THE UNIVERSITY OF MICHIGAN

Technical Report 32

CONCOMP

August 1970

THE MAD/I MANUAL

Bruce J. Bolas
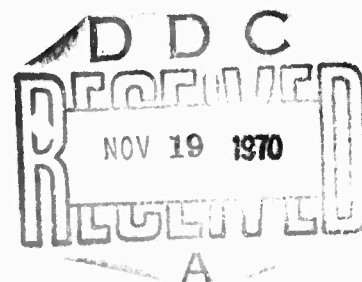
Allen L. Springer

Ronald J. Srodawa

197

DOCUMENT CONTROL DATA - R & D		
(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)		
1. ORIGINATING ACTIVITY *(Corporate author)* UNIVERSITY OF MICHIGAN CONCOMP PROJECT	2a. REPORT SECURITY CLASSIFICATION Unclassified	
	2b. GROUP	

3. REPORT TITLE

THE MAD/I MANUAL

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Technical Report 32

5. AUTHOR(S) *(First name, middle initial, last name)*

Bruce J. Bolas, Allen L. Springer, and Ronald J. Srodawa

6. REPORT DATE August 1970	7a. TOTAL NO. OF PAGES 184	7b. NO. OF REFS 0
8a. CONTRACT OR GRANT NO. DA-49-083 OSA-3050 b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) Technical Report 32	
	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DDC.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency

13. ABSTRACT

    This report contains the description of the MAD/I language and a user's guide for MAD/I in MTS (Michigan Terminal System).

    The pre-defined MAD/I language, described in this report, is a procedure-oriented algebraic language designed for general-purpose use. It is styled after such languages as ALGOL 60, MAD, and PL/I. MAD/I is also intended as a convenient "base" language for extension by a definitional facility. The language anticipates the definition (or re-definition) of: data types and structures, statements, operators and operations. The definitional facility itself is not described.

    The user's guide shows how to compile and run MAD/I programs in MTS. Sample runs are included. There is also a description of a compiler feature which allows assembler code for the IBM 360 to be compiled within a MAD/I program.

DD FORM **1473** 1 NOV 65

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
MAD/I programming languages extensible languages compilers						

T H E U N I V E R S I T Y O F M I C H I G A N

Technical Report 32

THE MAD/I MANUAL

Bruce J. Bolas

Allen L. Springer

Ronald J. Srodawa

Preface

        We use the term "MAD/I" to refer to any of four different
things:

  • The MAD/I Project -- a research project conducted at the
        University of Michigan Computing Center, and jointly
        sponsored by the Computing Center and the University's
        CONCOMP Project. (CONCOMP: Research in Conversational Use
        of Computers. Supported by the Advanced Research Projects
        Agency, Department of Defense, Washington, D.C.)

  • The MAD/I Facility -- a flexible translator-building facility
        which runs on the IBM System/360 computer. Created for the
        purpose of building the MAD/I Compiler, the MAD/I Facility
        provides for:

        (a) The definition of a user-specified programming
             language, subject to some constraints on lexicon,
             syntax, and interpretation sequence.

        (b) The specification in detail of a translation process
             for the defined language, using the MAD/I Facility as
             a "skeleton" for the translator.

        (c) The amalgamation of the translation specification with
             the skeleton, to produce a complete translator for the
             defined language.   In general, the resulting
             translator runs on the IBM 360, and directly produces
             object modules for the 360. The translator (and hence
             the language) can be modified ("extended") at compile
             time, producing an "extensible-language" effect.

  •   The   MAD/I   Language   --   a   particular   procedure-oriented
        algebraic language, designed for implementation on the
        MAD/I Facility. The MAD/I Language is intended to be
        useful both as a general-purpose language, and also as a
        convenient base or "core" language for extension into
        various dialects.

  • The MAD/I Compiler -- a compiler for the MAD/I Language,
        implemented in the MAD/I Facility. To date, the only
        version of the MAD/I Compiler runs in MTS (Michigan
        Terminal System) and produces object modules for MTS.

        This manual is the user's manual for the MAD/I Language and
the MAD/I Compiler.   It is intended as a reference manual
(rather than a teaching manual), and assumes that the reader is
already familiar with languages such as PL/I. The MAD/I
Language is described in Part I of this manual, and the Compiler

is described in Part II. There are also three appendices. The
reader is urged to read Section 1 (Introduction to the Language)
and Appendix A (Syntax Description Notation) first.

For further reference on MAD/I:

D. L. Mills, "The Syntactic Structure of MAD/I", CONCOMP
Technical Report 7, June 1968.

(Presents a formal syntactic description of an earlier
version of the MAD/I Language; also describes the
novel precedence-oriented parsing technique built into
the MAD/I Facility.)

Allen L. Springer, "Defaults and Block Structure in the
MAD/I Language", CONCOMP Memorandum 31, July 1970.

Ronald J. Srodawa, "An Example Definitional Facility in
MAD/I", CONCOMP Memorandum 32, July 1970.

The work presented here is the result of the combined
efforts of a number of people at the University of Michigan
Computing Center, working at various times over a period of five
years. The principal contributors are acknowledged below.

Professors Bruce W. Arden and Bernard A. Galler were the
project co-ordinators. They participated in the design of the
language, and wrote and edited earlier versions of the manual.

Most of the design work, and all of the programming and
debugging are due to:

> Bruce J. Bolas
> Charles F. Engle
> David L. Mills
> Allen L. Springer
> Ronald J. Srodawa
> Fred G. Swartz

Finally, we should like to express our appreciation to
Professors Robert C. F. Bartels (Director of the Computing
Center) and Franklin H. Westervelt (Director of the CONCOMP
Project), who have supported, encouraged, and sometimes prodded
the MAD/I effort since its inception.

Table of Contents

INTRODUCTION

MAD/I was originally conceived in 1965 at the University of Michigan Computing Center as a relatively simple carry-over of the MAD language from the IBM 7090 computer to the IBM System/360, with perhaps a few straightforward extensions. This goal, however, was later considerably revised. (For information on the MAD language, refer to: "The Michigan Algorithm Decoder", Revised Edition, 1966 (out of print); also see: B. W. Arden, B. A. Galler, and R. M. Graham, "The MAD Definition Facility", Communications of the ACM 12,8 (August 1969), 432-439.)

The CONCOMP Project was formed in December, 1965, to do extensive research in the conversational use of computers. CONCOMP needed a general-purpose language suitable for writing conversational programs, and also wanted facilities for defining new data types, operations, and statements into the language. Therefore, CONCOMP strongly supported the development of an extended MAD language which would serve these needs, and this became the new goal of the language project. In these early days, the language was known variously as "MADE", "COMET", or "MAD/360".

As work on the language and compiler progressed, it gradually became apparent that it was not feasible to retain useful compatibility with 7090 MAD. Also, the need for a flexible definitional facility forced the re-examination of basic concepts about the structure of programming languages. Eventually it was agreed that the MAD/I project was actually developing a new language (and compiler), which would be independent of MAD.

The goals of the MAD/I project were again re-defined. We now wanted a language-and-compiler system with the following features:

(A) It should contain a pre-defined algebraic language, suitable for conventional general-purpose use without any definitions from the user.

(B) The language should have a rather general syntax, so that a variety of new statements and operators might be defined into the same framework as the pre-defined constructs.

(C) It should contain a powerful definitional facility usable by a moderately sophisticated programmer. This facility should allow the user to modify the pre-defined language so as to satisfy his special requirements. In particular, it should allow the definition (or re-definition) of:

(1) Data structures and data types.

The MAD/I Manual

   (2) Statements (including declarations).

   (3) Operators and operations, either in terms of existing
       operations, or in terms of an assembler-like language
       allowing access to the object machine instruction set,
       at the user's option.

(D) The compiler should be reasonably fast, especially when the
    program contains no new definitions.

(E) The compiled object program should be reasonably efficient,
    although perhaps not highly optimized.

The earlier goal of compiling "conversational" programs was seen
as primarily an operating system problem.   This was nicely
fulfilled at Michigan by the development of MTS (Michigan
Terminal System), which also was partly supported by CONCOMP.

    The goals above have largely been fulfilled, with a few
exceptions. We will discuss them in order:

(A) The pre-defined MAD/I language is a useful general-purpose
    language. It has a syntactic structure somewhat like ALGOL
    60, but it includes many of the important features of MAD
    and PL/I.   The MAD/I Compiler has been working since late
    1968, and is being used for practical system programming
    work. Portions of the compiler itself have been written in
    MAD/I.

(B) The syntax rules of MAD/I are sufficiently general to allow
    a large "space" of possible definitions. A great variety
    of symbols, expressions, and statements is syntactically
    possible.

(C) The definitional facility exists, but it is not complete.
    The MAD/I language itself is implemented using this "MAD/I
    Facility", and one can indeed define new data types,
    statements, operators, etc. Unfortunately, this facility
    is too "low-level", and cannot be used without considerable
    study. A user-oriented facility is certainly feasible, but
    this requires more research and development.

(D) The compiler is unfortunately <u>not</u> fast.   It is large and
    very slow, because it constantly re-interprets definitions.
    With a little more work, the compiler could be speeded up
    by a factor of at least four.

(E) The object program now produced is reasonably efficient,
    although not highly optimized. Even better object code is
    possible.

PART I -- DESCRIPTION OF THE MAD/I LANGUAGE

Section 1: Introduction to the Language

1.1   General Features

This section briefly mentions some of the principal
concepts and features of the MAD/I language.

Input Form

The language is defined in terms of a continuous sequence
of characters, independent of card format or line boundaries.
The compiler accepts its input as a sequence of records (lines)
which may vary in length. This input is normally treated as
completely free-form, and is broken into a sequence of symbols.
Blanks and comments may be used freely between symbols, but most
symbols cannot contain blanks.

Symbols

In MAD/I two concepts have been separated: the form of a
symbol (how it is composed of characters), and the usage of the
symbol (how it functions as a language element). Examples of
symbol forms (called "lexical classes") are:

    Alphanumeric symbol   (e.g., F A32 BETA)

    Primed symbol   (e.g., 'IF' 'TRUE' 'END')

    Quoted symbol   (e.g., "CHAR-STRING" "001A4"X)

    Unsigned-integer symbol   (e.g., 4 003 5140)

    Special symbol   (e.g., + : , ( ) ¬=)

The symbols may be used in any of several ways; example usage
classes are:

    Identifier. Usually formed as an alphanumeric symbol, but
        the primed symbol 'DEFAULT' is also a pre-defined
        identifier.

    Keyword. The pre-defined keywords are primed symbols. An
        alphanumeric symbol (such as IF or BEGIN) could be
        defined as a keyword, but then it could not also be
        used as an identifier.

Part I -- Description of the MAD/I Language

Constant. The symbols 307, 'TRUE', 18.4E3, and "P=**" are all constant symbols.

Operator. The symbols + , = , .ABS. , .OR. , := , and ** are pre-defined operators.

Attributes and Declarations

Language items such as identifiers, constants, and expressions have <u>attributes.</u> Example attributes are:

Mode (e.g., 'INTEGER', 'CHARACTER', 'VARYING ARRAY', 'POINTER')

Storage layout (e.g., Length, Alignment, Dimension)

Storage class (e.g., Static, Automatic, Based)

Attributes of an item may be explicitly declared either in a declaration statement or by attaching a declaration to any occurrence of the item in the program. Declarations may appear anywhere in the program, and in particular need not precede the first occurrence of the item. There are also <u>default</u> attributes for items which are not completely declared. The defaults to be applied are themselves declarable.

Example declarations:

'INTEGER' I, K, N

'DECLARE' (ALPHA, BETA) 'FIXED ARRAY' (50,50) 'BOOLEAN'

I@('INTEGER') := N@('INTEGER') + 3

'DECLARE' 'DEFAULT' 'FLOATING LONG'

Expressions

A MAD/I expression is basically similar to an expression in FORTRAN, MAD, ALGOL, or PL/I, but is slightly more general. The four expressions

ALPHA , A + B , (X-Y)*Z , -XYZ

all have the usual meanings in the pre-defined language. However, the conventional concepts of "subscripted variable" and "assignment statement" are handled as expressions in MAD/I.

For example, if ALPHA is an array name, then ALPHA is considered a variable, but ALPHA(I) is <u>not</u> a variable; both

ALPHA and ALPHA(I) , however, are expressions (called "designators").   In "ALPHA(I)" the subscription operation is implied by the context of the array name expression followed by the left-parenthesis symbol; it is treated as a convenient way of writing "ALPHA .TAG. I", where .TAG. is the operator denoting subscription.

An assignment such as "AA := BB" is also an expression; its result is the same as AA , except that the value of AA has been set to the value of BB.   We could compute the maximum value of AA and BB with the statement:

        'IF° (MAX := AA) < BB , MAX := BB

The concepts of "operator" and "operation" have been separated.   For example, the special symbol + is pre-defined as an infix operator which, in the contexts of arithmetic operands, denotes "addition".   Addition is a binary (i.e., dyadic) operation. The + operator could, however, be defined to mean something other than addition for other contexts.

Statements

MAD/I statements are roughly similar to those of ALGOl 60 and PL/I.   There are five general statement classes: simple statements, compound statements, prefix statements, list statements, and declaration statements.

(1)   Simple statements.   A simple statement is either an expression or a "statement keyword" (reserved word) followed by a fixed number of expressions.

        A := BCD
        'GO TO' LABEL
        'ALLOCATE' STRUCT, K*10

(2)   Compound statements. A compound statement consists of a sequence of statements, separated by semicolons, and bracketed by a statement keyword and an "end keyword".

        'BEGIN' B := A; C := D 'END'

(3)   Prefix statements.   A prefix statement consists of a "prefix" followed by a "scope". The prefix consists of a statement keyword followed by a fixed number of expressions. The scope may consist either of one statement (separated from the prefix by a comma) , or of a sequence of statements separated by semicolons and terminated by an end keyword (separated from the prefix by a semicolon).

Part I -- Description of the MAD/I Language

```
'IF' A > 0 , B := A

'IF' A > 0 ; B := A ; C := D 'ENDIF'

'FOR' I:=1,1,I>N , G(I) := 0

'FOR' I:=1,1,I>N ;
      G(I) := 0 ; H(I) := 1 'ENDFOR'
```

(4) List statements. A list statement consists of a prefix followed by a varying number of expressions.

```
'READ' ('UNIT' 0), A, B, C

'PRESET' D := 1, F := 3.5, CH := "**"
```

(5) Declaration statements. These have two forms: the 'DECLARE' statement and the "inverted" declaration statements, as exemplified below.

```
'DECLARE' AA 'INTEGER', BB 'BOOLEAN',
      CC 'COMPONENT STRUCTURE' ('BIT'(8), 'BIT'(24))

'DCL' (DD, EE, FF) 'FLOATING', GG 'ENTRY NAME'

'INTEGER' II, JJ, XX

'FIXED ARRAY' A1(5,10), A2(4,4,4)
```

Statements to be successively executed are written in sequence, separated by semicolons. Empty statements are permitted. A statement may be labeled with an identifier, separated from the statement by a colon.

```
      I := 0;
LBL:  'IF' Z(I) ¬= 0, 'RETURN' I ;
      I := I + 1 ; 'GO TO' LBL
```

Embedded statements

Any statement (or sequence of statements) can be made to produce a result, and can therefore be used as an expression (i.e., "embedded" in an expression). The 'VALUE' prefix statement is provided for this purpose. The prefix designates a variable whose value at the end of the statement is used as the result of the statement. The 'VALUE' statement is enclosed in parentheses and used as an expression.

```
SUMSQUARE := ('VALUE' S ; S := 0 ; 'FOR' I:=1,1,I>N,
          S := S + (X(I) - Y(I))**2 'ENDVALUE')
```

Part I -- Description of the MAD/I Language

Program structure

MAD/I provides a "block structure" much like ALGOL 60 and PL/I. Each block is either a "compound-statement block" or a "procedure block". A compound-statement block has the form

    'BLOCK' • • • 'END'

where the • • • represents an arbitrary sequence of statements. Procedure blocks have several variations; they typically look either like

    'PROCEDURE' NAME.(PAR1,PAR2) ;
            • • •
    'END PROCEDURE'

or like

    'PROCEDURE' FN.(X,Y) := expression

Both kinds of blocks are statements, and can be used wherever a statement is valid. Blocks, therefore, may be nested. Block structure serves to delimit the scopes of declarations and names. Each block may either have its own default attributes, or may inherit the defaults of the enclosing block.

A MAD/I program is a block not contained in any other block. Each source program is separately compiled into its object program.

1.2 Introductory Examples

Let us suppose that X and Y are two arbitrary vectors in a vector space of 3 dimensions, and that we want a procedure which computes the Euclidean "distance" function between X and Y. The following program does this; the line numbers at the left margin are **not** part of the program.

```
01              'PROCEDURE' DIST.(X,Y);
02              'DCL' (X,Y) 'FIXED ARRAY'(3);
03              'INTEGER' I;
04      DIST:   SUM := 0. ;
05              'FOR' I := 1, 1, I > 3 ,
06                 SUM := SUM + ( X(I) - Y(I) ) ** 2 ;
07              'RETURN' SQRT.(SUM)
08              'END PROCEDURE'
```

The program is a procedure block; the procedure statement begins with the statement keyword 'PROCEDURE' and ends with the end keyword 'END PROCEDURE' in line 08.

Line 01 consists of the procedure prefix followed by a semicolon. The procedure prefix specifies that identifier DIST names an entry point of the procedure, and that identifiers X and Y are the formal parameters associated with that entry point. Since the prefix is followed by a semicolon, the rest of the 'PROCEDURE' statement will be a sequence of statements terminated by the end keyword 'END PROCEDURE'.

Line 02 consists of a 'DECLARE' statement followed by a semicolon. ('DECLARE' is abbreviated as 'DCL' -- many MAD/I keywords have abbreviations.) The statement specifies that X and Y are variables of 'FIXED ARRAY' mode, and that their values are arrays of 3 components, numbered from 1 to 3. 'FIXED ARRAY' means that the arrays have fixed dimensions; they cannot be re-dimensioned at run time. Since the mode of the array components is not explicitly declared, it is assumed to be the default mode; since the block contains no declaration for default mode, the pre-defined default of 'FLOATING SHORT' is used. Thus, the values of X and Y are arrays of 3 floating-point numbers. The semicolon at the end of line 02 is not part of the 'DECLARE' statement, but separates it from the next statement.

Line 03 contains a declaration statement which specifies that I is a variable of 'INTEGER' mode. This is called an "inverted" declaration statement, since it begins with an attribute keyword instead of 'DECLARE'.

Line 04 contains an "expression statement" labeled by the identifier DIST; this is the entry point of the procedure. The statement is an assignment expression, which sets the value of variable SUM to the floating-point value 0. SUM is not

explicitly declared, so it has the default mode 'FLOATING SHORT'.

Line 05 ['FOR' I := 1, 1, I > 3 ,] has the beginning of a 'FOR' statement, which specifies an iteration. The iteration variable is I; it is initialized to 1, and is incremented by 1 until the expression I > 3 is true. Since the 'FOR' statement prefix is followed by a comma, the scope of the iteration will be a single statement.

Line 06 [SUM := SUM + (X(I) - Y(I)) ** 2 ;] contains an expression statement, which is the statement repeatedly executed. The semicolon separates the 'FOR' statement and the 'RETURN' statement. The assignment expression increments the value of SUM by the square of the difference of the Ith components of the two vectors.

Line 07 ['RETURN' SQRT.(SUM)] contains a 'RETURN' statement. It evaluates the expression SQRT.(SUM) and returns the resulting value as the result of the DIST procedure. The identifier SQRT is implicitly declared to be 'ENTRY POINT' mode by its appearance as a procedure name in the procedure-call expression; since SQRT is not a label in this program, it is implicitly declared 'EXTERNAL' as well. Also, a procedure call on SQRT is assumed to produce a result of default mode. This program assumes that SQRT is an entry point of a (library) subroutine that computes the square root of a 'FLOATING SHORT' value and returns a result of the same mode. MAD/I itself does not have pre-defined procedures for the elementary functions.

Line 08 ['END PROCEDURE'] contains the 'END PROCEDURE' keyword which ends the procedure (and the program). We could also have used the general-purpose end keyword 'END' instead. Notice that no semicolon is needed between the 'RETURN' statement and the end keyword. Such a semicolon would do no harm, however; it would merely introduce an empty statement between the semicolon and the end keyword.

Part I -- Description of the MAD/I Language

As a second example, let us generalize the previous problem so that X and Y are vectors in a space of N dimensions, and that N is supplied as an actual parameter (argument) to the procedure. We could then re-write DIST as follows:

```
01          'PROC' DIST.(N,X,Y);
02          'DCL' (I,N) 'I', (X,Y) 'FA'(#) 'FS';
03          SUM := 0;
04          'FOR' I:=1,1,I>N,
05             SUM := SUM + (X(I)-Y(I))**2;
06          'RETURN' SUM ** 0.5 'END'
```

Line 01 is the same as before, except that 'PROCEDURE' is abbreviated as 'PROC', and N is added as a formal parameter.

Line 02 contains a single 'DECLARE' statement, which uses abbreviations. It declares that I and N have 'INTEGER LONG' mode, and that X and Y have 'FIXED ARRAY' mode with 'FLOATING SHORT' components. The special symbol # specifies that the array dimensions are to be obtained at run time from the actual parameters supplied for X and Y.

Line 03 is similar to line 04 before, except that the label DIST has been omitted, and the constant 0 has no decimal point. Since DIST is declared in the procedure prefix as an entry point, but DIST does not appear as a label, the entry point is considered to be at the first executable statement, which is "SUM := 0". The constant 0 has 'INTEGER LONG' mode, and will be converted to 'FLOATING SHORT' mode for assignment to SUM. The MAD/I compiler reserves the "right" to perform such a conversion at compile time.

Line 04 ['FOR' I:=1,1,I>N,] is the same as before, except that the iteration proceeds until the value of I exceeds the value of parameter N. If N is less than 1, then the iteration scope is never executed.

Line 05 [SUM := SUM+(X(I)-Y(I))**2;] is the same as before.

Line 06 ['RETURN' SUM ** 0.5 'END'] combines the functions of lines 07 and 08 before. Instead of explicitly calling a procedure SQRT, the MAD/I exponentiation operation is used. The 'END' keyword ends the program.

As a third example, we will re-write the generalized DIST procedure to use an "embedded statement":

```
01              'PROC' DIST.(N,X,Y) ;
02              'DCL' (I,N) 'I' ,  (X,Y) 'FA'(#)  'FS';
03    DIST:    'RETURN' ('VALUE' SUM := 0. , 'FOR' I:=1,1,I>N,
04              SUM := SUM+(X(I)-Y(I))**2)**0.5 'END'
```

Lines 01 and 02 are as before. Lines 03 and 04 contain a labeled 'RETURN' statement; the expression for the return value contains a parenthesized 'VALUE' statement.   The   'VALUE' statement prefix specifies the variable SUM and sets it to zero; the 'VALUE' statement scope is the 'FOR' statement, which is the same as before; the result of the statement is the value of SUM after the scope is executed. The 'VALUE' statement is enclosed in parentheses and its value raised to the 0.5 power. The 'END' keyword ends the program as before.

Part I -- Description of the MAD/I Language

Section 2: Symbols, Comments, and Spaces   (Lexical Structure)

2.0 Introduction

A source program in the MAD/I Language is a sequence of
characters -- letters, digits, blanks, and special characters.
A language processor must group successive characters together
into symbols, comments, and spaces. The resulting lexical
sequence of symbols constitutes the formal MAD/I program, and is
the only portion of the source program text that is of interest
to a compiler or interpreter. The comments, when included, are
solely for the convenience of human readers. Spaces serve to
separate symbols and comments; they have no other significance.

Because the MAD/I Facility is intended to be flexible, and
because the MAD/I Language design must allow for "extension" by
the user, the rules for forming and recognizing symbols have
been divorced from the uses (interpretations) of the symbols.
For example, in a typical "fixed" language, an identifier must
be formed as an alphanumeric symbol; in MAD/I, however, the user
can cause almost any symbol (e.g., a string of characters
enclosed in quotation marks) to be treated as an identifier.
There are also default interpretations for some symbol forms;
for example, an alphanumeric symbol not otherwise declared is
treated as an identifier.

2.1 Formation of Symbols   (Lexical Classes)

The rules for grouping characters together into symbols are
embedded in the lexical scanner of the MAD/I Facility; hence,
they are fixed. The lexical scanner recognizes ten general
categories (lexical classes) of symbols, which are listed here
and defined in detail below:

                1. Alphanumeric symbols
                2. Primed symbols
                3. Dotted symbols
                4. Quoted symbols
                5. Unsigned-integer symbols
                6. Unsigned-floating-point symbols
                7. Special symbols
                8. Percent symbols
                9. At-sign symbols
                10. Pound symbols

2.1.1 Alphanumeric Symbols

An alphanumeric symbol is a sequence of adjacent letters or digits, the first of which must be a letter. The "letters" are the upper-case characters A,B,...,Z, and the lower-case characters a,b,...,z. (It should be understood that these are 52 different characters.) The "digits" are the characters 0,1,2,...,9. An alphanumeric symbol must have at least one character, but no more than 256. Adjacent alphanumeric symbols must be separated by spaces or comments.

Usual usage: Identifiers
Default interpretation: Identifier

Examples: MADI
               X
               B90A2
               LongerSymbolThanMost

2.1.2 Primed Symbols

A primed symbol is a sequence of 1 to 254 letters, digits, or blanks, enclosed in "primes" (apostrophes, single-quote marks). All blanks between the primes are ignored, and are not considered as spaces.

Usual usage: Keywords, Constants

Examples: 'IF'
               'GO TO' , same as 'GOTO'
               'INTEGER'
               'DEFAULT'
               'TRUE'
               'NULL PT'

2.1.3 Dotted Symbols

A dotted symbol is a sequence of 1 to 254 letters or digits, the first two of which must be letters, enclosed in dots (periods). No blanks are permitted within a dotted symbol.

Usual usage: Operators

Examples: .A.
               .LS.
               .ASTYPEOF.
               .qq3.

2.1.4 Quoted Symbols

A quoted symbol is a sequence of zero or more characters enclosed in quotation marks (double-quote marks). A quoted symbol can also include a suffix character (X, P, or E) immediately after the closing quote (see below). Any characters, including blanks and special characters, can be written between the quotes; however, each occurrence of the quote (") character must be represented by two adjacent quotes (""). If a quoted symbol is followed by a symbol which begins with a quote or letter, the two symbols must be separated by a space (or comment). The four forms of quoted symbols are described below:

2.1.4.1 Character Symbols

A character symbol is a quoted symbol which has no suffix.

Usual usage: Character-string constants
Default interpretation: Constant of 'CHARACTER' mode; see
         Sec. 3.8.3.

Examples: "A"
               "** Error: IHC999 @ 51."
               """"   (contains one " character)

2.1.4.2 Hexadecimal Symbols

A hexadecimal symbol is a quoted symbol with the suffix character X . The characters between the quotes are restricted to the "hexadecimal digits": 0,1,...,9,A,B,C,D,E,F.

Usual usage: Constants
Default interpretation: Constant of 'INTEGER LONG' mode; see
         Sec. 3.8.4.

Examples: "A9E"X
               "BAD"X
               "2001940000"X

2.1.4.3   Pointer-Constant Symbols

A pointer-constant symbol is a quoted symbol with the suffix character   P .   The characters **between** the quotes constitute another symbol -- the identifier whose storage assignment will be "pointed to".

Usual usage: Pointer constants

Examples: "ALPHA"P
                "SIN"P

2.1.4.4   Entry-Name-Constant Symbols

An entry-name-constant symbol is a quoted symbol with the suffix character   E .   The characters **between** the quotes constitute another symbol -- the identifier (label) of the entry point to be "pointed to" by the entry-name constant.

Usual usage: Entry-name constants

Examples: "LAB12"E
                "SIN"E

2.1.5   Unsigned-Integer Symbols

An unsigned-integer symbol is a sequence of decimal digits, and is considered to be the usual decimal representation of a non-negative integer. Leading zeros are permitted, but commas and decimal points are not.

Usual usage: Integer constants
Default interpretation: Constant of 'INTEGER LONG' mode; see
          Sec. 3.8.1.

Examples: 38
                0
                00190

2.1.6   Unsigned-Floating-Point Symbols

An unsigned-floating-point symbol is a sequence of decimal digits, with either a single decimal point, or an "exponent part", or both. If the decimal point is written, it may be placed anywhere in the sequence of digits, and is interpreted according to the usual rules of decimal notation.   The decimal sequence may be suffixed by an "exponent part", which represents a multiplier value applied to the decimal number. The exponent part consists of the character   E   followed   by a decimal

integer, and represents a multiplier equal to 10 raised to the
power of the decimal integer.   The decimal integer may be
signed.

Usual usage: Floating-Point constants
Default interpretation: Constant of 'FLOATING SHORT' mode; see
            Sec. 3.8.2.

Examples: 1.57
               0.
               .1
               0.005
               10E3   (= 10 x 10^3 = 10^4)
             ⁻ 2.2E-07   (= 2.2 x 10^{-7})
               .04E+48   (= .04 x 10^{48})

2.1.7 Special Symbols

       The following special symbols are pre-defined in MAD/I;
they all have pre-defined interpretations as punctuation marks
and operators:

(	left-parenthesis
)	right-parenthesis
,	comma
;	semicolon
:	colon
...	ellipsis
#	pound-sign, number-sign
+	plus
-	minus
*	asterisk
/	slash
@	at-sign
.	dot, period
$	dollar-sign
¬	not-sign
&	ampersand
|	vertical bar
=	equal-sign, "equals"
<	less than
>	greater than
**	double-asterisk, power
:=	colon-equals, assignment, "gets"
¬=	not equal
<=	less than or equal
>=	greater than or equal
||	double-bar, concatenate

2.1.8 Percent Symbols

A percent symbol is a percent-sign immediately followed by a non-empty sequence of letters and digits. Percent symbols are used extensively in MAD/I as internal compiler symbols. Compiler-generated identifiers (such as the names of temporary results) are percent symbols. The programmer should avoid writing percent symbols unless he is deliberately using the low-level MAD/I Facility.

Examples: %TMP0007
                %A
                %MACRO

2.1.9 At-sign Symbols

An at-sign symbol is an at-sign followed by a non-empty sequence of letters and digits. At-sign symbols, like percent symbols, are used in MAD/I for internal compiler symbols. They are also used as component names (see Sec. 2.2.5). The programmer should avoid writing at-sign symbols unless he is writing a component name or deliberately using the low-level Facility. (Note: the single character ə is a special symbol, and is not classed as an at-sign symbol.)

Examples: əCLS
                əEX2
                əMODE

2.1.10 Pound Symbols

A pound symbol is a pound-sign followed by a non-empty sequence of letters and digits. Pound symbols are intended for use in the Compile-Time Facility, and are reserved as a class for that purpose. (Note: the single character # is a special symbol, and is not classed as a pound symbol.)

Examples: #COUNT
                #L12
                #ROWS

Part I -- Description of the MAD/I Language

2.2 Usage of Symbols   (Usage Classes)

Except for internal compiler symbols and special symbols which are punctuation marks, the MAD/I symbols can be categorized into five general usage classes, which are listed here and discussed in detail below:

1. Identifiers
2. Constants
3. Keywords
4. Operators
5. Component names

2.2.1 Identifiers

An identifier is a symbol used as a name of some data object such as an integer value, a pointer value, or a portion of a program. There are two kinds of identifiers: variables and labels.

There is also a special pre-defined identifier, the primed symbol 'DEFAULT'. This appears only in declaration statements, and is used as a controllable "prototype" for the assignment of default attributes (see Section 3).

2.2.1.1 Variables

A variable is an identifier used to name a data object (its "value"). The essence of a variable is that the particular data object named is not fixed, but may vary when the object program is executed ("run time"). For example, if the symbol K is a variable, it might (at run time) name first an integer value 15, and later an integer value -77, and still later an integer value 0 . A variable can also name a structured set of values, such as an array of floating-point values.

We remind the reader that computing machines do not manipulate abstract objects, such as numbers, directly. Rather, machines must manipulate concrete representations of such objects. Thus, when we say that the variable K names the integer value 15, we always mean that K names a finite representation of the integer 15, and that this representation is the value of K. With this distinction understood, we may say loosely that "K has the value 15", and hope there will be no confusion.

The computational properties of each variable are represented by the attributes assigned to the variable. An example attribute in MAD/I is mode, which characterizes both the range of values the variable can name, and the form of a typical

value. For example, a variable of 'INTEGER LONG' mode can only
name values which are integers encoded (in System/360) as
fullword (32-bit) fixed-point binary numbers.   Another   example
mode is 'FIXED ARRAY', which specifies that the variable names
an <u>array</u> of values, that the bounds on each dimension of the
array are fixed,   that all the values in the array are of the
same mode, and that the values are located at regularly-spaced
intervals in computer storage.   In this case an individual value
is designated by writing <u>subscripts</u> after the variable.   For
example, if AR is a variable of 'FIXED ARRAY' mode, then a value
in the array may be designated by an expression such as AR(1)   .
<u>Note:</u> AR(1) is <u>not</u> a variable, but is an expression called a
<u>designator</u> (see Section 4).

2.2.1.2 Labels

A label is an identifier that names a <u>fixed</u> object.   Unlike
a variable, the value of a label cannot change at run time;
thus, a label is a kind of constant. Labels are used   only to
name statements in programs; each label is written in front of
the statement it names, separated from the statement by a   colon
(:).   In the pre-defined language, there are only two modes a
label can have: 'TRANSFER POINT' mode and 'ENTRY POINT' mode
(see Section 3).

Part I -- Description of the MAD/I Language

2.2.2   Constants

A constant is a symbol (or @-expression -- see below) which denotes a **fixed** value. The value of each constant is computed in advance of run time, and may or may not be explicitly represented in the object module. A constant may have either of two forms:

(1) A single "constant symbol".

    Examples: 419
                   23.7E-3
                   'TRUE'

(2) A constant followed by the @ symbol followed by a parenthesized declaration; i.e., an @-expression whose left operand is a constant.

    Examples: 419@('INTEGER SHORT')
                   "4E000000"X@('FLOATING SHORT')
                   "ABC"@('CHARACTER'(8))@('ALIGN'(8))

The pre-defined constant symbols include:

    Unsigned-integer symbols   (Sec. 2.1.5)
    Unsigned-floating-point symbols   (Sec. 2.1.6)
    Character symbols   (Sec. 2.1.4.1)
    Hexadecimal symbols   (Sec. 2.1.4.2)
    Pointer-constant symbols   (Sec. 2.1.4.3)
    Entry-name-constant symbols   (Sec. 2.1.4.4)
    The Boolean constants 'TRUE' and 'FALSE'
    The character constant 'NULL C'
    The varying-character constant 'NULL VC'
    The pointer constant 'NULL PT'
    The entry-name constant 'NULL EN'

The reader will note that **signed** constants have not been mentioned. The application of a + or - prefix symbol to a constant results in an **expression** which is not called a "constant", although it is constant-valued.

2.2.3 Keywords

A keyword is a symbol which has been assigned a particular
use in a MAD/I statement form. All keywords, both pre-defined
and user-defined, are reserved symbols.   The pre-defined
keywords are all primed symbols, and can be roughly divided into
four informal categories: statement keywords, end keywords,
phrase keywords, and attribute keywords.

Statement keywords are those which begin and identify a
statement form.   Each occurrence of a statement keyword is
considered to begin a statement of the form identified by the
keyword (see Section 5).

Examples: 'PROCEDURE'
                                 'IF'
                                 'FOR'
                                 'GO TO'

End keywords are those which end statements.   An end
keyword is part of the statement it ends, and is the last symbol
of the statement (see Section 5).

Examples: 'END'
                                 'END PROCEDURE'
                                 'ENDIF'

Phrase keywords are those which separate expressions, or
identify optional expressions, within a larger statement
context. Some phrase keywords are used like commas -- to
separate expressions.   Others are prefix keywords which combine
with an expression to form a larger expression (see Section 5).

Examples: 'WITH'
                                 'TO'
                                 'END OF FILE'
                                 'SAVE CODE'

Attribute keywords are those which represent attributes,
such as mode and storage class, and are used to declare the
attributes of identifiers and expressions.   Attribute keywords
normally appear as suffix or infix keywords within declarations,
but they can also function as statement keywords in the
"inverted" declaration form (see Sections 3 and 5.9).

Examples: 'FLOATING LONG'
                                 'ENTRY POINT'
                                 'NOT NEW'
                                 'EXTERNAL'

2.2.4 Operators

An operator is a symbol which denotes an operation on data objects. The same operator may denote a number of different operations; the appropriate operation for each occurrence of the operator is determined by the context of that occurrence.

Each occurrence of an operator has one or two adjoining expressions which denote the operands (data objects) of that occurrence. Each operator is in exactly one of four syntactic categories:

A prefix operator is written before its operand expression.

A postfix operator is written after its operand expression.

An infix-left operator is written between its operand expressions; infix-left operators of equal precedence associate left-to-right (see Section 4).

An infix-right operator is written between its operand expressions; infix-right operators of equal precedence associate right-to-left.

Note: In order to preserve both the above distinction and traditional notation, two pre-defined symbols get special treatment: Whenever the minus (-) symbol appears in the context of a prefix operator, it is transformed to the negation (.NEG.) operator. Whenever the plus (+) symbol appears in the context of a prefix operator, it is dropped and ignored. Thus, the plus and minus signs retain their usual dual roles.

All pre-defined operators are either special symbols or dotted symbols. They and their associated pre-defined operations are discussed in Section 4.

Examples: +   (infix-left)
                       ¬   (prefix)
                       :=   (infix-right)
                       .ABS.   (prefix)
                       .REM.   (infix-left)

2.2.5 Component Names

A component name is a symbol used to name (or label) a component of a structured data object. All component names are established at compile time, through their use in declarations of structured variables.

For example, the declaration statement

'DCL' CMPLXN 'CS' (@REAL 'FS', @IMAG 'FS')

declares that CMPLXN is a variable of 'COMPONENT STRUCTURE' mode (see Sec. 3.1.2.2) with two components; each component has 'FLOATING SHORT' mode. Also, the symbols @REAL and @IMAG are declared to be component names, which name (for the variable CMPLXN) the first and second components respectively. The first component of CMPLXN can then be designated by the expression CMPLXN $ @REAL , and the second component by CMPLXN $ @IMAG .

The same component name can be used for different structured variables, and can name different components of those variables.

The compiler currently requires that all component names must be at-sign symbols, in order to distinguish them from identifiers. This restriction may be relaxed in the future. Also, the compiler presently allows component names which are at-sign symbols to be written like ordinary subscripts; e.g., CMPLXN(@REAL) and CMPLXN(@IMAG).

2.3   Comments and Spaces

Any source program text may be enclosed in "comment delimiters" to form a comment. Comment delimiters are the character pairs << and >> . Thus, the following is a comment:

<center><<THIS IS A COMMENT.>></center>

Once a left comment delimiter (<<) is recognized, all characters after it are considered part of the comment until the first right comment delimiter (>>) occurs. Comments must not be nested. Comments may be inserted at any point in the text of the program except within symbols. They are bypassed in the initial scan of the text, and they have no effect on the object program.

Spaces are sequences of one or more adjacent blank characters which are not embedded within a symbol or comment. Spaces are significant in that they will separate symbols which would otherwise "run together". Blank characters within a primed symbol, a quoted symbol, or a comment are legal and are not considered as spaces; blanks cannot be embedded in any other symbols.

Section 3: Attributes

3.0   Introduction to Attributes

Attributes are simply "significant properties". That is, the attributes of an item in a MAD/I program are those properties of the item which are of interest to the language processor (beyond the purely syntactic properties, which are not considered attributes). Attributes must be determined by the language processor, at "compile time", in order to produce a correct translation of the program. The "items" for which attributes are defined include identifiers, constants, and expressions, as follows:

Each identifier has attributes that characterize the values that it names and the scope of the identifier itself. Every identifier acquires the following attributes:

A mode, which specifies both the possible values of the identifier and the representation form of a value. The mode may be either a primitive mode or a structured mode. A primitive mode (such as 'BOOLEAN' mode) describes a relatively simple data object and requires no other mode for its definition. A structured mode (such as 'COMPONENT STRUCTURE' mode) describes a "structured" object which has components (or produces results) which have their own modes.

A scope, which is that portion of the program over which the identifier is uniquely "defined"; i.e., that portion in which another occurrence of the same symbol is another occurrence of the same identifier.

A storage class, which specifies the manner in which storage is associated with the identifier.

If the identifier is a variable, then it also acquires at least two "storage layout" attributes:

A length, which specifies the amount of storage (number of bytes) required for a value.

An alignment (alignment factor), which specifies a constraint on the position (in storage) of the storage associated with the identifier.

Storage layout attributes do not apply to labels.

Part I -- Description of the MAD/I Language

If the identifier has a structured mode, then that includes
additional attribute information to describe its value; for
example, a fixed array has "dimension", and its components have
a mode.

Each <u>constant</u> has a mode, a length, an alignment, and a
storage class (which is always 'STATIC' -- see Sec. 3.4.1). It
also has a <u>value,</u> of course, but this is not considered an
"attribute".

Each <u>expression</u> has a mode, which is the mode of its
result. It may also have a storage class and storage layout
attributes.

Most of the attributes are represented in the language by
attribute keywords, which are used in declarations to specify
attributes of items. Some attribute keywords take "suffixes",
which may be optional or required, to specify additional
attribute information.

Sections 3.1 to 3.4 below describe the various attributes
themselves in detail. Sections 3.5 to 3.9 describe the various
ways of assigning attributes to identifiers, constants, and
expressions.

3.1 Mode Attributes

Every identifier, constant, and expression acquires a mode
attribute, either by explicit declaration or by implicit
declaration. Each mode characterizes a set of possible values,
and also the form of a value of that mode in computer storage.
In general, the mode of an item strongly affects the treatment
of that item by the operators and statements of the language.

Most modes also carry implied values for the length and
alignment attributes, so that these often need not be explicitly
declared. For examples: 'CHARACTER' mode has an implied
alignment of 1, and 'FLOATING LONG' mode has implied length 8
and implied alignment 8.

There are two classes of modes in MAD/I -- primitive modes
and structured modes:

The primitive modes characterize relatively simple data
objects, and are "atomic" in the sense that they require no
other modes for their definition. Most of the primitive modes
(like 'INTEGER SHORT' mode) are intentionally defined as direct
counterparts to the hardware data types of the IBM System/360.

> Note: This approach allows the MAD/I user strong
> control over the machine code produced by the
> compiler. Thus, it enhances the usefulness of MAD/I
> for writing system programs for the IBM 360. However,
> this approach also has the disadvantage that it tends
> to make programs machine-dependent and thus less
> transferable.

Some of the primitive modes are called "arithmetic" modes. This
simply means that they characterize arithmetic values -- i.e.,
representations of numbers -- and that some arithmetic
operations (such as addition) have been pre-defined for them.

The structured modes characterize relatively complex
objects which have "components" or "results" for which more mode
information may be required. For example, if an item has 'FIXED
ARRAY' mode, then the mode of the components of the array must
somehow be determined. This can be explicitly declared by a
declaration statement such as

'DECLARE' A 'FIXED ARRAY' (7) 'POINTER' 'BOOLEAN'

which declares that the value of variable A is a fixed array of
7 components, each of which is a 'POINTER' mode value pointing
to an object of 'BOOLEAN' mode. 'FIXED ARRAY' and 'POINTER' are
structured modes, while 'BOOLEAN' is a primitive mode.

Structured modes are also very useful for creating new, complex, user-defined modes. This will be discussed more fully in Section 9.

The pre-defined modes are listed below, and defined in detail in the following subsections:

<u>Primitive modes:</u>   <u>3.1.1</u>

    'INTEGER SHORT' mode   3.1.1.1
    'INTEGER LONG' mode   3.1.1.2
    'FLOATING SHORT' mode   3.1.1.3
    'FLOATING LONG' mode   3.1.1.4
    'PACKED' mode   3.1.1.5
    'BIT' mode   3.1.1.6
    'BOOLEAN' mode   3.1.1.7
    'CHARACTER' mode   3.1.1.8
    'VARYING CHARACTER' mode   3.1.1.9
    'FILE NAME' mode   3.1.1.10
    'TRANSFER POINT' mode   3.1.1.11

<u>Structured modes:</u>   <u>3.1.2</u>

    Array modes   3.1.2.1
      'FIXED ARRAY' mode   3.1.2.1.1
      'VARYING ARRAY' mode   3.1.2.1.2
    'COMPONENT STRUCTURE' mode   3.1.2.2
    'ALTERNATE' mode   3.1.2.3
    'POINTER' mode   3.1.2.4
    'ENTRY POINT' mode   3.1.2.5
    'ENTRY NAME' mode   3.1.2.6

3.1.1   Primitive Modes

3.1.1.1   'INTEGER SHORT' mode

'INTEGER SHORT' mode (abbreviation 'IS') is an arithmetic mode with integer values ranging from -32768 (-2^{15}) to +32767 ($2^{15}-1$). It has implied length 2 and implied alignment 2.

3.1.1.2   'INTEGER LONG' mode

'INTEGER LONG' mode (abbreviations 'IL', 'INTEGER', 'I') is an arithmetic mode with integer values ranging from -2147483648 (-2^{31}) to +2147483647 ($2^{31}-1$). It has implied length 4 and alignment 4.

3.1.1.3   'FLOATING SHORT' mode

'FLOATING SHORT' mode (abbreviations 'FS', 'FLOATING', 'F') is an arithmetic mode with signed (+ or -) values whose magnitudes range from about 5.4×10^{-79} ($1/16 \times 16^{-64}$) to about 7×10^{75} ($(1-16^{-6}) \times 16^{63}$), and with a maximum precision of six hexadecimal digits (about seven decimal digits). The zero value is also included. This mode has implied length 4 and alignment 4.

3.1.1.4   'FLOATING LONG' mode

'FLOATING LONG' mode (abbreviation 'FL') is an arithmetic mode with essentially the same range of values as 'FLOATING SHORT' mode, but with a maximum precision of 14 hexadecimal digits (about 17 decimal digits). It has implied length 8 and alignment 8.

3.1.1.5   'PACKED' mode

'PACKED' mode is an arithmetic mode with integral values expressed as signed decimal integers. The attribute keyword 'PACKED' takes an optional suffix of the form (L) , where L specifies the length attribute, and must be a constant from 1 to 16. If the suffix is omitted, the default length is 1. The value is 2xL-1 decimal digits, with a sign. An alignment of 1 is implied.

3.1.1.6   'BIT' mode

A 'BIT' mode (no abbreviation) value is a fixed-length
string of bits, which can also be treated as an unsigned binary
integer. The attribute keyword 'BIT' takes an optional suffix
of the form (L) , where L is an integer constant from 1 to 32
which specifies the bit length of the string. If the suffix is
omitted, the default length is 1.

The compiler currently requires that the storage assigned
to each 'BIT' mode item lie within a single 32-bit word (4 bytes
with alignment 4); that is, 'BIT' mode storage assignments
cannot overlap word boundaries. Thus, the alignment of a 'BIT'
mode item is determined by two special rules:

(a) If the item is a component of a 'COMPONENT
     STRUCTURE', it is aligned to the next available
     bit, unless the item will then not fit within
     that word, in which case it is aligned to the
     first bit in the next word.

(b) In all other cases, the exact alignment is
     undefined. For this reason, 'BIT' mode items
     currently should not be passed as parameters,
     except as components of component structure or
     array parameters.

3.1.1.7   'BOOLEAN' mode

'BOOLEAN' mode (abbreviation 'BOOL') has exactly two
values: 'TRUE' and 'FALSE'. It has implied length 1 and
alignment 1.

3.1.1.8   'CHARACTER' mode

A 'CHARACTER' mode (abbreviation 'C') value is a fixed-
length string of characters.   The attribute keyword takes an
optional suffix of the form (L) , where L is an integer constant
between 1 and 256 which specifies the number of characters in
the string. If the suffix is omitted, the default length is 1.
Since each character requires one byte of storage, the length
attribute is the same as the character length.   The implied
alignment is 1.

3.1.1.9   'VARYING CHARACTER' mode

A 'VARYING CHARACTER' mode (abbreviation 'VC') value is a varying-length string of characters, together with an integer value which specifies the current length of the string. The attribute keyword takes an optional suffix of the form (L) , where L is an integer constant from 1 to 32767 which specifies the maximum string length. If the suffix is omitted, the default maximum length is 256. At run time, the string value may be any sequence of characters whose length does not exceed the maximum length. This includes the "null" string, which has length zero. The implied alignment is 2, and the implied length is 2+(the maximum length). The constant symbol 'NULL VC' is a pre-defined constant of this mode; it has maximum length zero, string length zero, and length attribute 2.

3.1.1.10   'FILE NAME' mode

A value of 'FILE NAME' mode (no abbreviation) is a set of specifications for a MAD/I file. It has implied length 4 and alignment 4. Refer to Section 6 (Input/Output).

3.1.1.11   'TRANSFER POINT' mode

An item of 'TRANSFER POINT' mode (no abbreviation) names a point in the program which can receive a transfer of control from elsewhere within the same program, but which does not have the special properties of an "entry point". 'TRANSFER POINT' mode is never explicitly declared; instead, identifiers are contextually declared as labels by appearing before a colon in front of a statement. As long as nothing in the program causes a label to be declared as 'ENTRY POINT' mode, then it will receive 'TRANSFER POINT' mode by default. All items of this mode have 'STATIC' storage class. Items of 'TRANSFER POINT' mode cannot be formal parameters, nor can they be passed as actual parameters.

3.1.2 Structured Modes

Structured modes characterize data objects which involve other, "subordinate" data objects. We will use the general term "subtype" to talk about a subordinate data object (such as a "component" or "result") of a structured-mode object. Unless otherwise stated, a subtype may be of any mode, including the structured modes.

3.1.2.1 Array modes

A value of an array mode ('FIXED ARRAY' or 'VARYING ARRAY') is an array of one or more component values. An array is a "homogeneous" structure in that all its components share the same mode, storage class, and storage layout attributes. The attribute keyword takes an obligatory suffix -- a parenthesized list of subscript bounds specifications (see Appendix A for explanation of syntax notation):

        array-suffix = (list , bounds)

        bounds = [integer ...] integer

        integer = [+ | -] unsigned-integer-symbol

If two integers are given, the first one specifies the lowest value (lower bound) for that subscript position, and the second specifies the highest value (upper bound). If only one integer is given, it specifies the upper bound, and the lower bound is assumed to be 1. The upper bound must be greater than or equal to the lower bound. The number of "bounds" given specifies the number of dimensions of the array and also the number of subscripts which must be given to designate a component. This number, the set of bounds values, and the spacing (in storage) of components, together constitute the dimension attribute of the array. Dimension is classed as a "storage layout" attribute.

The array-suffix may be followed by an optional explicit declaration of a typical array component. If this is omitted, the current 'DEFAULT' declaration is copied as an implicit declaration. The storage class of a component cannot be declared; it is always the same as the storage class of the array.

Example:

        'DECLARE' A 'FIXED ARRAY'(10,-2...5) 'CHARACTER'(5)

declares that variable A names a two-dimensional array with 10 "rows" (first subscript) numbered from 1 to 10, and 8 "columns"

(last subscript) numbered from -2 to 5. The array has 10 x 8 = 80 components, each of which is a fixed-length string of 5 characters.

The components of an array are assigned storage at regularly-spaced intervals. The minimum distance from the beginning of one component to the beginning of the next is the "aligned length" of a component, which is computed as the length of the component, extended as needed to satisfy the alignment of the next component. Along each dimension (subscript position) of an array, the successive components have the same spacing, which is always a multiple of the aligned length. The default alignment of an array is the same as the alignment of its components.

When the components of an array must be treated in serial order (as in storage assignment or in I/O transmission), some sort of 'sequencing rule" must be employed. The default array sequencing rule is called row-major order, and is the order produced by varying each subscript from its lower bound to its upper bound, the last subscript varying first, then the next-to-last, etc., until all combinations have been produced. For example, if we have declared A 'FIXED ARRAY' (-1...1,2,0...2) , then row-major order gives the sequence: A(-1,1,0), A(-1,1,1), A(-1,1,2), A(-1,2,0), A(-1,2,1), A(-1,2,2), A(0,1,0), A(0,1,1), — , A(0,2,2) , A(1,1,0), — , A(1,2,2) .

3.1.2.1.1   'FIXED ARRAY' mode

'FIXED ARRAY' mode (abbreviation 'FA') characterizes arrays whose dimension attributes are permanently fixed at compile time. That is, the number of dimensions, the subscript bounds, and the spacing of components are all declared just once, and cannot vary at run time. The MAD/I translator can take advantage of this invariance to make operations on fixed arrays more efficient than the same operations on varying arrays.

3.1.2.1.2   'VARYING ARRAY' mode

'VARYING ARRAY' (abbreviation 'VA') characterizes arrays whose dimension attributes can vary at run time. The number of dimensions of a varying array is fixed, but the subscript bounds and the spacing of components can be varied dynamically with the 'REDIMENSION' statement (see Section 5.14). The dimension attribute declared in the program controls both the storage allocated to the array, and also the interpretation of any 'PRESET' assignments into the array. The re-dimension operation will not vary the location or size of the storage allocated to the array. For such arrays the declared dimensions should be large enough to accommodate the maximum-size array anticipated.

3.1.2.2 'COMPONENT STRUCTURE' mode

A value of 'COMPONENT STRUCTURE' mode (abbreviation 'CS') is a structure of component values which may be of different modes. Thus, a component structure is a "non-homogeneous" structure, in that its components need not all share the same mode and storage layout attributes. A component structure is a single compact data object in storage, so all its components do share the same storage class attribute. The attribute keyword takes an obligatory suffix -- a parenthesized list of component declarations:

        cs-suffix = (list , component-decln)

        component-decln =
                [component-name] declaration-string

Each component-decln declares one component, which may have any mode, primitive or structured (except 'TRANSFER POINT' and 'ENTRY POINT' modes), as specified by the declaration-string. If a component-name (see Sec. 2.2.5) is given, then its interpretation for the particular component structure is a name for the component being declared. If the declaration-string is empty, then the current 'DEFAULT' declaration applies to that component.

For example, the declaration statement

        'DCL' AGG 'CS'('BIT'(8), 'INTEGERSHORT', 'POINTER')

declares that AGG is a variable of 'COMPONENT STRUCTURE' mode, with three unnamed components. The first component has 'BIT' mode, the second has 'INTEGER SHORT' mode, and the third has 'POINTER' mode. Since the components are not named, they can be designated only by their ordinal position; e.g., the second component must be designated by the expression AGG(2) .

As another example, the declaration

        'DCL' VCHAR 'CS' ('IS', @LNG 'IS', @CHS 'FA'(50)'C')

declares that variable VCHAR is a component structure with three components: two of 'INTEGER SHORT' mode, and one 'FIXED ARRAY' whose components are single characters. The first component can only be designated as VCHAR(1) ; the second as either VCHAR(2) or VCHAR$@LNG ; the third as either VCHAR(3) or VCHAR$@CHS . The Ith character of the third component may be designated as either VCHAR(3)(I) or VCHAR$@CHS(I) .

The components of a component structure have the same ordering in storage as in the structure declaration. Each component is positioned after the preceding component, with the

minimum gap needed to satisfy its alignment attribute. The
default alignment of the structure is the maximum of the
individual component alignment attributes. The default length
of the structure is the minimum length needed to contain all the
aligned components.

3.1.2.3   'ALTERNATE' mode

A value of 'ALTERNATE' mode (abbreviation 'ALT') is similar
to a component structure (Sec. 3.1.2.2), except that the
"components" are actually alternative interpretations of the
value itself. It is equivalent to a component structure in
which the components all overlap each other, instead of being
disjoint. The attribute keyword takes the same form of
obligatory suffix, a cs-suffix.

For example, the declaration statement

'DCL' WHAT 'ALTERNATE' ('INTEGER','FLOATING')

declares that variable WHAT has 'ALTERNATE' mode, with two
interpretations: WHAT(1) has 'INTEGER' mode, and WHAT(2) has
'FLOATING' mode. We could also have used named components.

The value of an 'ALTERNATE' mode item has only one
component mode at a time, and it is the programmer's
responsibility to know which it is at any given point in the
program. Dynamic mode testing is not provided.

The alignment of the "structure" is the maximum of its
component alignments, and its length is the maximum of its
component lengths.

3.1.2.4   'POINTER' mode

A value of 'POINTER' mode (abbreviation 'PT') is a pointer
to another value. A "pointer" is the MAD/I counterpart of a
computer storage address, but is not necessarily implemented as
a simple address. The attribute keyword takes an optional
suffix, which must be a declaration-string, to describe the
value pointed to. If the suffix is omitted, the usual default
is not applied; rather, the value pointed to is considered as
"not declared". 'POINTER' mode has implied length 4 and
alignment 4.

Examples:

        'DCL' P1 'POINTER' 'INTEGER'

declares variable P1 to have 'POINTER' mode, with values that
point to values of 'INTEGER' mode.

        'DCL' P2 'PT' 'PT'

declares that the value of P2 is a pointer to a pointer to a
"not declared" value.

This mode has a pre-defined constant, 'NULL PT' , whose
value is a "null" pointer; it does not point to a value. Other
pointer constants may be defined as described in Sections
2.1.4.3 and 4.2.10.

3.1.2.5   'ENTRY POINT' mode

An item of 'ENTRY POINT' mode (abbreviation 'EP') names a
point in some program which can receive a transfer of control,
and which has the special properties of an "entry point"
described below. The attribute keyword takes an optional
suffix, which must be a declaration-string, to describe the
value produced as a result of "calling" the designated entry
point. If the suffix is omitted, the 'DEFAULT' declaration is
applied. Every entry point has the following properties:

(1) It can receive either "go to" or "call" transfers of
control.

(2) It can receive transfers ("call" or "go to") from
external procedures as well as procedures within the
same program.

As a consequence, an entry point is more "expensive" than an
ordinary transfer point, since it must perform whatever rituals
are required by program linkage conventions. Also, some entry
points take parameters, whereas transfer points cannot.

An item may be declared 'ENTRY POINT' in several ways:

(1) Explicitly, with the 'ENTRY POINT' keyword.

(2) Contextually, as an identifier in a procedure-prefix.
For example,

'PROCEDURE' F.(X), G.(Y,Z)

contextually declares F and G as 'ENTRY POINT' .

(3) Implicitly, as a label which is declared 'ACCESSIBLE' .

(4) Contextually, as an identifier G which occurs in either
the context G.(—) or the context "G"E , and has not
been explicitly declared in the block containing the
occurrence. In this case, if G is not a label in the
program, it is contextually declared 'EXTERNAL' as
well.

Any entry point declared only contextually is assumed to produce
result values of default mode.

An item declared 'ENTRY POINT' must be an identifier.
Thus, structured values cannot have components or results of
'ENTRY POINT' mode. Also, every entry point must have 'STATIC'
storage class.

Part I -- Description of the MAD/I Language

3.1.2.6   'ENTRY NAME' mode

A value of 'ENTRY NAME' mode (abbreviation 'EN') is a
pointer to an entry point, together with additional information
to determine an environment for the entry point. The attribute
keyword takes an optional suffix, which must be a declaration-
string, to describe the value produced as a result of calling
the entry point pointed to. If the suffix is omitted, the
'DEFAULT' declaration is applied.

Items of 'ENTRY NAME' mode can be constants, variables, or
expressions.   (Note that 'TRANSFER POINT' and 'ENTRY POINT'
modes do not have variables.)   This mode has a pre-defined
constant, 'NULL EN' , whose value is a "null" entry name; it
does not specify an entry point or an environment. Other entry
name constants may be defined as described in Sections 2.1.4.4
and 4.2.10. Their values point to entry points, but they do not
carry environment information; this is filled in when the value
is used.

Unlike entry points, entry names are not restricted to
static storage class. Also, entry names may be passed as actual
parameters to procedures, and may be used in 'RETURN TO'
statements.

3.2   Storage Layout Attributes

Storage layout attributes are applicable to items which are variables, constants, or expressions. These attributes determine both the amount of storage allocated to an item, and the arrangement of the item's value in the allocated storage.

3.2.1   Length attribute

The "length" attribute of an item specifies the amount of storage allocated to that item. The attribute keyword 'LENGTH' takes an obligatory suffix of the form (L) , where L is a non-negative integer constant which specifies the length in bytes. The length attribute is taken as the maximum of the value of L and the implied length (if any) implied by other attributes of the item.

For example, 'INTEGER LONG' mode has an implied length of 4 bytes. The declaration statement:

```
'DECLARE' A 'INTEGER',
          B 'LENGTH'(2) 'INTEGER',
          C 'LENGTH'(6) 'INTEGER'
```

would cause variables A, B, and C to receive length attributes of 4, 4, and 6 bytes, respectively.

For those modes whose keywords do not take "length" suffixes, a declaration of the length attribute has no effect on the value of the item, but only allows extra storage to be allocated. In the above example, variable C will get six bytes of storage, but its values will still be the 4-byte integers of 'INTEGER' mode.

3.2.2   Alignment attribute

The alignment attribute of an item specifies a constraint on the positioning of its allocated storage. The attribute keyword 'ALIGN' takes an obligatory suffix of the form (A) , where A is an integer constant which specifies the "alignment factor" for the desired alignment. The only valid values for A are 1, 2, 4, and 8. The alignment attribute for the item is taken as the maximum of the value of A and the alignment (if any) implied by other attributes of the item. The values 1, 2, 4, and 8 correspond to byte, halfword, fullword, and doubleword alignments, respectively.

For example, the declaration

        'DCL' HH 'ALIGN'(4) 'CHARACTER'(8)

gives variable HH an alignment of 4.

3.2.3   Dimension attribute

    The dimension attribute applies only to items of array
modes (see Sec. 3.1.2.1), and specifies the number of dimensions
of the array, the subscript bounds for each dimension, and the
spacing (in storage) of the components along each dimension.
This attribute does not have its own keyword, but is declared as
part of the mode declaration.

3.3 Scope Attributes

Unlike the other attributes discussed so far, scope attributes are concerned with names rather than values. Scope attributes apply only to names which are identifiers, and represent properties of the identifier itself. Scope attributes are closely related to the block structure of programs, and to the "renaming convention" which allows the use of the same symbol as different names in different contexts. These concepts are discussed in Section 7.

There are two scope attributes: "scope" and "owner", defined loosely as follows:

The scope of a name is that portion of a program (or set of programs) in which the name is uniquely "known". The scope of a name always includes the program text internal to the block in which the name is declared (explicitly or implicitly), and always excludes the scope of any other name represented by the same symbol.

The owner of a name is that block which provides the storage associated with the name. This is significant for external names, whose scopes extend to more than one program.

Neither of the scope attributes has a direct attribute keyword. Instead, MAD/I has a combination of language conventions and scope-controlling keywords, which allow the user precise control of these attributes. The keywords are: 'NEW', 'NOTNEW', 'GLOBAL', 'EXTERNAL', and 'ACCESSIBLE'; they are applied like ordinary attribute keywords, and are described below.

When a name is explicitly declared, it is normally considered to be "new" to the "current block" (the smallest block enclosing the declaration). It is thus a new name, its scope is limited to the current block, and the current block is its owner. But this is not always true for contextual declarations; see Sec. 3.7.1.

'NOTNEW' specifies that the name declared is not new to the current block. This keyword causes the scope and ownership of the name to be extended to the next outer block, as though all declarations for that name (except the 'NOTNEW' declaration) in the current block were written in the next outer block instead.

'GLOBAL' specifies " 'NOTNEW' all the way out". A name declared 'GLOBAL' has its scope extended out to all blocks containing the declaration, in the same manner as for 'NOTNEW'.

If 'NOTNEW' or 'NEW' is declared for the special identifier 'DEFAULT', the declaration affects not the scope of 'DEFAULT'

itself, but rather all names in the current block which are used
but not explicitly or contextually declared. Such names require
a default assumption about their scope, and this is controlled
by 'DEFAULT'. If 'DEFAULT' is declared 'NEW' in the current
block, then such names are considered new to the block; if
'DEFAULT' is declared 'NOTNEW', then such names are not
considered new to the block, and thus are known in the next
outer block. If neither 'NEW' nor 'NOTNEW' is declared for
'DEFAULT', the action is as if 'NOTNEW' were declared.

'FXTERNAL' (abbreviation 'EXT') specifies that the name is
an "external name", that it has static storage class, and that
its owner is not in the current program (the program containing
the 'EXTERNAL' declaration). The scope of the name is extended
from the current block to outside the program.

'ACCESSIBLE' (abbreviation 'ACC') specifies that the name
is an external name, that it has static storage class, and that
its owner is in the current program. The scope of the name is
extended from the current block to outside the program. The
same name must not also be declared 'EXTERNAL' in the same
program, since this would cause conflicting declarations of its
owner.

If two or more external names are represented by the same
symbol, they are considered as one name whose scope is the union
of the individual scopes. This rule is applied when programs
are linked together, and allows the scope of a name to extend to
multiple programs. Ultimately, at run time, some program must
be the unique owner of the name.

In a 'PROCEDURE' block, the names of all entry points are
contextually declared 'NOTNEW', so that the entry points are
known outside the procedure itself. If the procedure is the
outermost block, these names are contextually declared
'ACCESSIBLE', and thus become external names.

3.4   Storage Class Attributes

Every identifier, constant, and expression has exactly one storage class attribute, which specifies the manner in which storage is associated with the item. The storage class may be declared either explicitly or implicitly.

When an item is associated with storage, then we say that storage is <u>allocated</u> for the item. The storage may be allocated either "statically" (before run time) or "dynamically" (during run time). Since storage is used primarily to contain values, the value of an item is not defined unless storage is allocated for the item.

The storage classes are: static, automatic, based, and formal parameter. The <u>default</u> storage class is <u>static.</u>

3.4.1   Static storage class

This attribute has the keyword 'STATIC'. It specifies that storage for the declared item is allocated before run time, and cannot be de-allocated or re-allocated during run time.

Static storage class is required for external names, and is therefore explicitly declared by the 'EXTERNAL' and 'ACCESSIBLE' keywords.

3.4.2   Automatic storage class

This attribute has the keyword 'AUTOMATIC'. It specifies that storage for the declared item is allocated during run time, whenever the block which owns the item is activated. The storage is de-allocated when the block is terminated. During the block activation, the storage cannot be re-allocated.

3.4.3   Based storage class

This attribute has the keyword 'BASED'. It specifies that storage for the declared item is dynamically allocated and de-allocated during run time, under explicit control of the program. Storage for based variables may be allocated and de-allocated in either of two ways:

(1) With the .ALLOC. operator and a pointer-valued expression. (See Section 4.)

(2) With the 'ALLOCATE' and 'DEALLOCATE' statements. (See Sec. 5.13.)

Part I -- Description of the MAD/I Language

3.4.4   Formal parameter storage class

This attribute has no keyword. Instead, it is contextually
declared for variables which appear as "formal parameters" in
procedure prefixes (see Sec. 5.7). Formal parameter storage
class is a consequence of the "call by reference" convention of
MAD/I. It specifies that storage for the declared item is
dynamically allocated when the formal parameter is "bound" to
its corresponding actual parameter (argument). This "binding"
occurs whenever the procedure is activated through an entry
point for which the formal parameter is declared. See Section
5.7 for more information.

3.5 Attribute Assignment -- Introduction

Sections 3.5 to 3.9 describe declarations, which specify attributes of items. The items declared may be identifiers, constants, or expressions.

Declarations may be explicitly written by the programmer; these are called explicit declarations. Also, the language processor may "infer" attributes which have not been explicitly indicated, but which are implied by the program and the rules of the language; the inference of an implied attribute is called an implicit declaration.

Implicit declarations may arise in two ways, by "context" or by "default":

The appearance of an item in a certain context can constitute a contextual declaration for the item.

If an item lacks some necessary attribute, which has been neither explicitly nor contextually declared, then it may receive a "default" attribute by default declaration.

A declaration may have either "unconditional" or "conditional" effect; that is, its application to the item may be unconditional, or may depend upon the absence of prior declarations. In general, explicit declarations are all unconditional, default declarations are all conditional, and contextual declarations can be either. For each item in the program, its attributes are assigned in the following order:

(1) Assign attributes specified unconditionally. These attributes must not conflict; if they do, it is an error.

(2) Assign attributes specified conditionally by contextual declarations, wherever their conditions are satisfied.

(3) Assign default attributes, wherever needed.

Constants and expressions normally do not require explicit declarations, since a constant gets default attributes determined by its lexical class, and an expression gets default attributes determined by its operator and operands. However, the programmer can explicitly control each constant with the @ operator and each expression with the .ASTYPE. and .ASTYPEOF. operators described in Section 3.9.

The default attributes for variables can differ from block to block. They are themselves declarable (See Sec. 3.7.2).

Part I -- Description of the MAD/I Language

3.6 Explicit Declarations

There are three forms of explicit declarations:

(1) The 'DECLARE' statement form.
(2) The "inverted" declaration statement form.
(3) The @-expression form.

The three forms are closely related; they are simply alternative
ways of writing declarations. Therefore, much of the syntax of
(explicit) declarations is common to all three forms. This
syntax is described below, both in prose and in syntax notation.
More information on @-class operators will be found in Section
4.2; the 'DECLARE' and 'DECLARE DEFAULT' statements are also
treated in Section 5.9.

Every explicit declaration requires: an occurrence of the
item being declared, and a "declaration string" of attribute
keywords and their suffixes. We will defer, for the moment, the
declaration of items which are constants or expressions, and
focus on the declaration of identifiers. The identifiers to be
declared by a given declaration string are written as an
"identifier list", which may be either a single identifier, or a
parenthesized list of identifiers:

        identifier-list = identifier | (list , identifier)

Examples: A
                     (B,C,BETA)
                     (X)

The declaration string will be applied individually to each
identifier in the list.

A "declaration string" (decln-string) is a sequence of
attribute keywords and their suffixes. The sequence is
interpreted from left-to-right, and there is a restriction on
the ordering of the keywords: any keyword which appears after a
structured-mode keyword applies to a "subtype", and must be
legal for that usage. (We will use the general term "subtype"
to talk about a subordinate data object (such as a component) of
a structured mode.) Each keyword, together with whatever suffix
it has, specifies a "declaration" about the identifier or a
subtype. At most one mode declaration may appear for the
identifier itself.

Syntax of decln-string:

    decln-string = [list non-mode-decln] [mode-decln]

    non-mode-decln = scope-decln | storage-class-decln
                | storage-layout-decln

    scope-decln = 'NEW' | 'NOTNEW' | 'GLOBAL'
              | EXTERNAL' | 'ACCESSIBLE'

    storage-class-decln = 'STATIC' | 'AUTOMATIC' | 'BASED'

    storage-layout-decln = 'LENGTH' (integer)
             | 'ALIGN' (integer)

    integer = [+ | -] unsigned-integer-symbol

    mode-decln = primitive-mode-decln | structured-mode-decln

    primitive-mode-decln =
        'INTEGER SHORT' | 'INTEGER LONG'
        | 'FLOATING SHORT' | 'FLOATING LONG'
        | 'PACKED' [(integer)]
        | 'BIT' [(integer)]
        | 'BOOLEAN'
        | 'CHARACTER' [(integer)]
        | 'VARYING CHARACTER' [(integer)]
        | 'FILE NAME'

    structured-mode-decln =
        'FIXED ARRAY' (list , bounds) subtype-decln
        | 'VARYING ARRAY' (list , bounds) subtype-decln
        | 'COMPONENT STRUCTURE' (list , component-decln)
        | 'ALTERNATE' (list , component-decln)
        | 'POINTER' [subtype-decln]
        | 'ENTRY POINT' subtype-decln
        | 'ENTRY NAME' subtype-decln

    bounds = [integer ...] integer

    subtype-decln = [list storage-layout-decln] [mode-decln]

    component-decln = [component-name] subtype-decln

Examples of decln-string:

(a)   'INTEGER SHORT'
(b)   'PACKED' (7)
(c)   'EXTERNAL'
(d)   'BASED' 'ALIGN' (8)
(e)   'NOTNEW' 'CHARACTER'
(f)   'FIXED ARRAY' (20;

(g)   'FIXED ARRAY' (5,10) 'LENGTH' (4) 'BIT' (18)
(h)   'BASED' 'ALTERNATE' ('INTEGER', 'POINTER' 'BOOL')
(i)   'ALIGN' (8) 'POINTER'

In example (g) above, the 'LENGTH' declaration follows the
structured-mode keyword, and thus it applies to the <u>components</u>
of the array, rather than to the array itself. In example (i),
the 'ALIGN' declaration applies to the pointer value itself,
rather than the object pointed to. Note that scope and storage
class can not be declared for subtypes; therefore the following
are examples of <u>invalid</u> declaration strings:

        'FIXED ARRAY' (3) 'INTEGER' 'BASED'
        'POINTER' 'EXTERNAL' 'FLOATING'

    Having defined and illustrated identifier lists and
declaration strings, we will now describe the various forms of
explicit declarations.

3.6.1 The 'DECLARE' statement form

The 'DECLARE' statement (abbreviation 'DCL') is the "root" form for explicit declarations. It consists of the statement keyword 'DECLARE' followed by one or more identifier lists, each followed by a declaration string, separated by commas.

DECLARE-statement =

'DECLARE' list , { identifier-list decln-string }

Note that a decln-string can be "empty"; i.e., it can be omitted. The effect of the 'DECLARE' statement is:

(a) Each identifier in each identifier list is "declared" in the current block. This will usually cause it to be "new" to the current block (see Section 3.3).

(b) In each identifier list, each identifier receives the attribute specifications defined by the declaration string (if any) immediately following. These attributes are specified "unconditionally".

Examples of 'DECLARE' statements:

```
'DECLARE' A
'DECLARE' (B,C,D,E)
'DECLARE' M,NN,P
'DECLARE' AA 'INTEGER'
'DCL' (BB,CC,DD) 'BOOLEAN'
'DCL' FF 'FIXED ARRAY' (0...5) 'FLOATING',
    (GG,HH) 'BASED' 'INTEGER',
    FLAGS 'ACCESSIBLE' 'BIT' (32)
```

3.6.2 Inverted declaration statement form

The "inverted" declaration statement form is provided
solely for programmer convenience. It may be considered as a
"transformation" of the 'DECLARE' form, in which the 'DECLARE'
keyword is replaced by an attribute keyword, which is extracted
from the declaration string. Some example pairs of equivalent
statements:

        'DECLARE' A 'INTEGER'
        'INTEGER' A

        'DECLARE' B 'FIXED ARRAY' (4,4) 'FLOATING'
        'FIXED ARRAY' B (4,4) 'FLOATING'

        'DCL' (C,D,E) 'CHARACTER' (50), F 'CHARACTER' (5)
        'CHARACTER' (C,D,E) (50), F (5)

Each inverted declaration statement consists of an
attribute keyword (which also functions here as a statement
keyword) followed by one or more identifier lists separated by
commas; each identifier list may be followed by whatever suffix
the keyword needs, followed by the remainder of the desired
declaration string.

        inverted-declaration-statement =
            attribute-keyword list , (identifier-list
                            [decln-suffix] [decln-string])

        decln-suffix = (integer)
                     | (list , bounds) [subtype-decln]
                     | (list , component-decln)
                     | subtype-decln

The inverted statement is treated as though it were transformed
to a 'DECLARE' statement by replacing the initial attribute
keyword with 'DECLARE', and inserting the attribute keyword
immediately after each identifier list.

More examples of inverted declaration statements:

        'INTEGER' A,B,C

        'LENGTH' S (10), T (8) 'INTEGER', U (8) 'CHARACTER'

        'BASED' (L,M,N) 'INTEGER', (P,Q) 'BOOLEAN',
            STR 'COMPONENT STRUCTURE' ('BIT' (8), 'BIT' (24))

3.6.3 The ∂-expression form

The ∂-expression declaration form is included primarily for specifying the attributes of constants, but it may also be used for identifiers. This form allows an explicit declaration to be attached to an ordinary occurrence of an identifier or constant in an expression. The declaration consists of the item being declared, followed by the infix operator ∂ , followed by a parenthesized declaration string:

∂-expression = {identifier | constant} ∂ (decln-string)

The effect of the ∂-expression for an identifier is the same as that of a 'DECLARE' statement with the same identifier and decln-string. The effect for a constant will be discussed in Section 3.8.   The result of the expression is the same as the result of the identifier or constant.

Examples: ABC ∂ ('INTEGER')
                     17 ∂ ('PACKED'(5))
                     "00C1C2C3" ∂ ('CHARACTER')

3.7 Implicit Declarations

3.7.1 Contextual Declarations

Contextual declarations are those which are implied by the usage of items in certain particular contexts. An appearance of an item in one of these contexts constitutes a contextual declaration about the item. The following contexts are defined:

Statement label. If an identifier appears before the special symbol : in front of a (possibly empty) statement, that identifier is contextually declared as a label , and as "new" to the current block.

Procedure-prefix entry point. Each identifier which appears as an entry point in a proc-prefix (Sec. 5.7) is contextually declared as:

    (a) a label;
    (b) "notnew" to the procedure block;
    (c) of 'ENTRYPOINT' mode.

If the procedure block is the outermost block, each such identifier is contextually declared 'ACCESSIBLE' as well.

Procedure-prefix formal parameter. Each identifier which appears as a formal parameter in a proc-prefix is contextually declared as a variable, "new" to the procedure block, and of formal parameter storage class.

Procedure-call. If an identifier appears as the left operand of the procedure-call operator (.), it is contextually declared as 'EXTERNAL' and 'ENTRYPOINT'. This is specified "conditionally", so that if any explicit declarations appear for the identifier, the contextual declaration will not be applied. The scope of the identifier is otherwise not affected. If the identifier appears as a label in the program, the 'EXTERNAL' specification is not applied.

3.7.2 Default Declarations

MAD/I does not require that the attributes of each
identifier be declared completely. For example, if a variable
is declared to have an array mode, but the mode of the array
components is not explicitly declared, this is not an error. In
each program block there is a set of <u>default</u> attributes, which
are used to "fill-in" attributes which have been neither
explicitly nor contextually declared. There are default
attributes for storage class and mode, and a default rule for
determining the scope attribute. In each block, the default
information for that block is associated with the special
identifier 'DEFAULT' , which is itself declarable as described
below.

The rules for applying default attributes to a given
<u>variable</u> in a given block are:

(1)   If the variable has been used in the block, but has not
      been explicitly or contextually declared, then its scope
      with respect to this block is determined from 'DEFAULT' as
      follows:

      (a)   If 'DEFAULT' is declared 'NEW' or 'NOTNEW' in
            this block, the variable is "new" or "notnew" to the
            block, respectively (see Sec. 3.3).

      (b)   If 'DEFAULT' is <u>not</u> declared either 'NEW' or
            'NOTNEW' in this block, the variable is "notnew" --
            the usual case.

(2)   If the variable has no storage class specified, apply the
      default storage class. This may be any storage class other
      than formal parameter.

(3)   If the variable has no mode specified, apply the default
      mode. If the variable has a structured mode specified, but
      some subtype (e.g., component, result) mode is not
      specified, then apply the default mode to each such
      subtype.

The rule for applying default attributes to a given <u>label</u>
in a given block is: if the label has no mode specified, apply
'TRANSFER POINT' mode; if the label has 'ENTRY POINT' mode but
the subtype mode is not specified, apply the default mode to the
subtype.

The default information itself is declarable for each
block. It can be explicitly declared in any of three ways:

(1)   With the 'DECLARE DEFAULT' statement (abbreviation 'DCLD';
      see Section 5.9).

      E.g., 'DECLARE DEFAULT' 'INTEGER LONG'

(2)   With a 'DECLARE' statement with 'DEFAULT' in an identifier
      list.

      E.g., 'DECLARE' 'DEFAULT' 'AUTOMATIC' 'FLOATING'

(3)   With an inverted declaration statement with 'DEFAULT' in an
      identifier list.

      E.g., 'FIXED ARRAY' 'DEFAULT'(3) 'FLOATING'

If the default information for a block is not completely
explicitly declared, then the missing attributes are "filled in"
from the defaults of the next outer block. For this purpose,
the outermost program block is considered as contained in an
imaginary block with defaults 'STATIC' 'FLOATING SHORT'. For
example, if the outermost program block contained the
declaration

        'DECLARE DEFAULT' 'FIXED ARRAY'(3)

and no other explicit declaration of defaults, then the defaults
for that block would be

      Storage-class: 'STATIC'
      Mode: 'FIXED ARRAY'(3) 'FLOATING SHORT' .

3.8 Attributes of Constants

As previously described in Sections 2.1 and 2.2.2, constants have various external forms, which we call "lexical classes". For each constant, the compiler must be able to compute an appropriate internal form for computation. MAD/I allows the explicit specification of attributes of constants, and provides that the conversions from external to internal forms are controlled by both lexical class and additional attributes.

For each lexical class of constant symbol, there is a standard conversion to a specific mode. The programmer can use the ∂ operator to declare additional attributes of an occurrence of a constant symbol. The 'LENGTH' and 'ALIGN' attributes can be used to adjust the storage allocation and positioning of the internal form. For some lexical classes, the mode attribute is also declarable. All constants have only 'STATIC' storage class. The rules for the various lexical classes are described below. The conversion rules themselves are not declarable, nor are they affected by the defaults established for identifiers.

3.8.1 Unsigned-integer symbols

Standard conversion: 'INTEGER LONG' mode.

Alternate conversions: 'INTEGER SHORT', 'FLOATING SHORT', 'FLOATING LONG', 'PACKED' (with optional length).

Example: 305∂('IS') converted to 'INTEGER SHORT'.

3.8.2 Unsigned-floating-point symbols

Standard conversion: 'FLOATING SHORT' mode.

Alternate conversion: 'FLOATING LONG'.

Example: 12.37∂('FL') converted to 'FLOATING LONG'.

3.8.3 Character symbols

Standard conversion: 'CHARACTER' mode, with length equal to the number of characters represented between the quotes.

Alternate conversion: 'CHARACTER' mode, with length greater than that implied by the symbol; the internal form is extended on the right with character-fill characters (blanks).

Part I -- Description of the MAD/I Language

Example: "ABCDE"@('CHARACTER'(8))

3.8.4 Hexadecimal symbols

Standard conversion: 'INTEGER LONG' mode; the hexadecimal
     digits are treated as an integer expressed in base 16.

Alternate conversions: 'INTEGER SHORT' mode: base 16 integer.
     'PACKED' mode: base 16 integer. 'CHARACTER' mode: the
     hexadecimal digits are treated as a bit string, and are
     left-justified in the storage allocated for the constant,
     with trailing zero bits as fills.   'FLOATING SHORT' and
     'FLOATING LONG' modes: bit string left-justified, with
     trailing zero bits as fills.

Example: "01FF"X@('C'(5)) converted to 'CHARACTER' mode.

3.8.5 Pointer-constant symbols

Standard conversion: 'POINTER' mode.

Alternate conversions: none.

3.8.6 Entry-name constant symbols

Standard conversion: 'ENTRY NAME' mode.

Alternate conversions: none.

3.9 Attributes of Expressions

Most expressions need not have their attributes explicitly
declared.   Instead, an expression's attributes are implicitly
"synthesized" from the attributes of its operands, according to
a "mode context" rule of its operator.   But sometimes the
implied attributes cannot be synthesized because of incomplete
information (e.g., a pointer value may point to an "undeclared"
value). Also, a programmer may occasionally need to "override"
the   implied   attributes.   Thus, there are two pre-defined
operators which allow the programmer to explicitly declare
attributes   of   expressions;   these   are   the .ASTYPE.   and
.ASTYPEOF. operators, described below.

.ASTYPE. (abbreviation .AS.) is an infix operator which
takes an expression as its left operand and a parenthesized
declaration string as its right operand:

astype-expression = expression .ASTYPE. (decln-string)

The result of the astype-expression is exactly the result of
"expression", but with the mode and storage-layout attributes
specified by "decln-string"; the storage class of the result is
always the storage class of "expression".

For example, suppose we wish to create a "translate table"
of characters, such that for each integer which is the internal
code of a character, the table maps that integer to the
'CHARACTER' mode value which has that internal code. Thus, the
table defines an "identity" translation on character codes. Let
the table be named TTC; it might be constructed for the IBM 360
by the following program segment:

```
'DCL' I 'INTEGER', IB 'BIT'(8),
    TTC 'FIXED ARRAY'(0...255) 'CHARACTER'(1);

'FOR' I:=0,1,I>255;  IB := I;
    TTC(I) := IB .ASTYPE. ('C') 'ENDFOR'
```

This example assumes (correctly) that the length and alignment
of a variable declared 'BIT'(8) will satisfy the requirements of
'CHARACTER'(1). All uses of .ASTYPE. and .ASTYPEOF.   involve
such assumptions; it is the programmer's responsibility to be
sure they are correct.

.ASTYPEOF. is an infix operator which takes an expression as its left operand and a parenthesized variable as its right operand:

    astypeof-expression = expression .ASTYPEOF. (variable)

The result of the astypeof-expression is exactly the result of "expression", except that its mode and storage-layout attributes are copied from "variable".

For example, in the "translate table" example described above, we could have written:

    'DCL' I 'INTEGER', IB 'BIT'(8), CHAR 'C'(1),
        TTC 'FIXED ARRAY' (0...255) 'C';

    'FOR' I:=0,1,I>255; IB := I;
        TTC(I) := IE .ASTYPEOF. (CHAR) 'ENDFOR'

Section 4: Expressions

4.0 Basic Concepts

An _expression_ is a syntactic form which specifies the
computation of a _result._ An expression can be a "primitive
expression" (such as a constant or identifier), whose result
requires little or no computation, or a "composite expression"
(such as A+(B*C), VT(I), or .ABS.X), whose result is obtained
from an _operation_ upon the result(s) of one or more sub-
expressions, or an "embedded statement", which is described
later.   Each composite expression consists of an _operator,_ with
one or two adjoining "operand expressions" whose results are the
operands for the operation. The operation itself is determined
by the operator, together with selected attributes (such as
mode) of the operand expressions.

The result of an expression is either a _reference_ or a
value. A "reference" is, in effect, a "location" -- an
identification of a region of storage which contains a value
(primitive or structured). An expression which produces a
reference result is called a _designator._

Note: A "reference" is not the same as a "pointer". A pointer
     is a type of value which corresponds to a reference, but
     which can be copied and otherwise manipulated.

Example expressions	Designator?
ALPHA	Yes
VECT(I,J)	Yes
AA + BB	No
FN.(X)	No
V := 1	Yes
"ABC"	No
-10	No
.IND. PTR	Yes
A ** .ABS.B	No
(A+B)/(C-D)	No

Any expression can be enclosed in parentheses without
affecting its meaning. Parentheses so used act as "grouping
marks" only, and do not convert an expression into a "list" or
"sequence".

The operators, besides being categorized as prefix,
postfix, infix-left, and infix-right (see Sec. 2.2.4), are also
assigned _precedences_ ("precedence levels", "priorities",
"binding strengths"). Operator precedences are used in the
usual way to resolve the structure of expressions which are not
fully parenthesized, and which might otherwise be syntactically

ambiguous.   See Sec. 4.3 for the precedences of the   pre-defined
operators.

The order of computation of a composite expression is   only
defined   as   constrained   by   the   structure   of the expression,
together   with   the   interpretation   rules   of   the   individual
operations.   In particular, we do not say that an expression is
normally   evaluated   "left-to-right".   For   example,   in   the
expression   (A+B)*(C+D),   the   sub-expressions   A+B and C+D must
both be evaluated before   the   *   operation,   but   they   may   be
evaluated in either order.

Expressions   are   used   (syntactically)   to   build   statements.
That is, an expression can constitute a statement or a part of a
statement.   Likewise, it is possible to use statements to   build
expressions.   Any   MAD/I   statement   can   be   made   into   a
parenthesized statement which produces   a   well-defined   result;
such   statements   are   called   "embedded   statements",   and they
qualify as expressions.   See Sec.   5.6 for more   information   on
embedded statements.

We will occasionally wish to talk about   expressions   which
produce   results   of   certain   modes.   We   will   use   the   term
"arithmetic   expression"   to   refer   to   any   expression   which
produces   a   result   of   an   arithmetic   mode: 'INTEGER SHORT',
'INTEGER LONG', 'FLOATING SHORT', 'FLOATING LONG', or   'PACKED'.
We will also use the term "character-string expression" to refer
to any expression producing a result of 'CHARACTER' or   'VARYING
CHARACTER'   mode.   Similarly,   "arithmetic   designator"   and
"character-string designator" refer to arithmetic and character-
string expressions which produce reference results.

4.1 Primitive Expressions

There   are   only   two   kinds   of   primitive   expressions:
identifiers and constants.

An identifier (a variable or a label) produces a   reference
result -- a reference to the storage currently allocated for the
identifier,   which   is   assumed   to   contain   the   value   of   the
identifier.   If   no   storage   is so allocated, this is an error
condition, and the result is undefined.

A constant produces a value result -- the value denoted   by
the   constant.   A   constant   may   or   may   not   be   explicitly
represented in the   object   module;   it   may   or   may   not   have
associated storage.

4.2 Operations

The various pre-defined operations are listed below.   For
each operation there is a pre-defined operator which denotes
that operation in some contexts. The contexts are all defined,
unless otherwise indicated, by the mode attributes of the
operand expressions. Thus, for each operation and corresponding
operator, we give those pre-defined "mode contexts" for which
the operator denotes the operation and the operation is defined.
We also give the mode and type of the result.

Legend for context tables:

The various 1st operand modes label the rows, and the
various 2nd operand modes label the columns. Each row-
column position corresponds to a potential mode context for
the operator. Each blank position defines an invalid mode
context; each non-blank position defines a valid mode
context. A non-blank table entry has one of two forms: (1)
A mode abbreviation of one or two letters, meaning that the
operation is defined for this context, and the result has
the mode indicated. (2) A digit (1 or 2) followed by a
mode abbreviation, meaning that a copy of the 1st or 2nd
operand (as indicated by the digit) is converted to the
mode indicated, and the table is re-entered with the new
mode context.

4.2.1 Arithmetic operations

The arithmetic operations are primarily defined on the
following "arithmetic modes":

Keyword	(Abbrev. For Tables)
'INTEGER SHORT'	IS
'INTEGER LONG'	IL
'FLOATING SHORT'	FS
'FLOATING LONG'	FL
'PACKED'	PK

Some arithmetic operations are also defined for some contexts
using the "semi-arithmetic" modes:

'BIT'	BT
'POINTER'	PT

The arithmetic operations are as follows:

Addition (binary), denoted by "+"; e.g., "A + B". The operand-result contexts are summarized in the table below.

"+"	FL	FS	IL	IS	PK	BT	PT
FL	FL	2FL	2FL	2FL	2FL		1IL
FS	1FL	FS	2FS	2FS	2FS		1IL
IL	1FL	1FS	IL	2IL	2IL	2IL	PT
IS	1FL	1FS	1IL	IS	2IS	2IS	PT
PK	1FL	1FS	1IL	1IS	PK		1IL
BT			1IL	1IS		BT	
PT	2IL	2IL	PT	PT	2IL		

Subtraction (binary), denoted by "-"; e.g., "A - B". The result is a value -- the value of the 1st operand minus the value of the 2nd operand. See the following table.

"-"	FL	FS	IL	IS	PK	BT	PT
FL	FL	2FL	2FL	2FL	2FL		
FS	1FL	FS	2FS	2FS	2FS		
IL	1FL	1FS	IL	2IL	2IL	2IL	
IS	1FL	1FS	1IL	IS	2IS	2IS	
PK	1FL	1FS	1IL	1IS	PK		
BT			1IL	1IS		BT	
PT	2IL	2IL	PT	PT	2IL		IL

Multiplication (binary), denoted by "*"; e.g., "A * B". The result is a value -- the product of the operand values. See the following table.

"*"	FL	FS	IL	IS	PK	BT
FL	FL	2FL	2FL	2FL	2FL	
FS	1FL	FS	2FS	2FS	2FS	
IL	1FL	1FS	IL	2IL	2IL	2IL
IS	1FL	1FS	1IL	IS	2IS	2IS
PK	1FL	1FS	1IL	1IS	PK	
BT			1IL	1IS		BT

Division (binary), denoted by "/"; e.g., "A / B". The result is a value -- the quotient obtained by dividing the 1st operand value by the 2nd operand value. If both operands have integer-like (not floating-point) modes, the operation is "integer division". See the table for multiplication, above.

Remainder (binary), denoted by ".REM."; e.g., "I .REM. J". The result is a value -- the remainder obtained from dividing the 1st operand value by the 2nd operand value. See the following table.

".REM."	IL	IS	PK	BT
IL	IL	2IL	2IL	2IL
IS	1IL	IS	2IS	2IS
PK	1IL	1IS	PK	
BT	1IL	1IS		BT

Negation (unary), denoted by ".NEG." (or prefix "-"); e.g., ".NEG. A", "-A". The result is a value -- the arithmetic negative of the operand value. See the following table.

".NEG."	FL	FS	IL	IS	PK
	FL	FS	IL	IS	PK

Absolute value (unary), denoted by ".ABS."; e.g., ".ABS. A". The result is a value -- the value of the operand if that is non-negative, otherwise its negative. See the table for negation, above.

Exponentiation (binary), denoted by "**"; e.g., "A ** B". The result is a value -- the 1st operand value raised to the power of the 2nd operand value. See the following table.

"**"	FL	FS	IL	IS	PK	BT
FL	FL	2FL	FL	2IL	2IL	2IL
FS	1FL	FS	FS	2IL	2IL	2IL
IL	1FL	1FS	IL	2IL	2IL	2IL
IS	1FL	1FS	1IL	IL	2IS	2IS
PK	1FL	1FS	1IL	1IS	IL	
BT			1IL	1IS		BT

4.2.2 Relational operations

The six pre-defined relational operations are described as
a group. The result of each is a Boolean value -- representing
whether the operand values sati:fy the specified relation.

Equality (binary), denoted by "="; e.g., "A = B".

Inequality (binary), denoted by "¬=" and ".NE."; e.g., "A ¬= B",
    "A. NE. B".

Greater-than (binary), denoted by ">"; e.g., "A > B".

Greater-than-or-equal-to (binary), denoted by   ">=";   e.g.,
    "A >= B".

Less-than (binary), denoted by "<"; e.g., "A < B".

Less-than-or-equal-to (binary), denoted by "<="; e.g., "A <= B".

All six operations are defined for the mode contexts shown
in the following two tables:

REL'N	FL	FS	IL	IS	PK	BT
FL	BL	2FL	2FL	2FL	2FL	
FS	1FL	BL	2FS	2FS	2FS	
IL	1FL	1FS	BL	2IL	2IL	2IL
IS	1FL	1FS	1IL	BL	2IS	2IS
PK	1FL	1FS	1IL	1IS	BL	
BT			1IL	1IS		BL

REL'N	BL	BT	VC	C
BL	BL	2BL		
BT	1BL	BL		
VC			BL	BL
C			BL	BL

Boolean values are compared by interpreting 'TRUE' as "1"
and 'FALSE' as "0". Character strings are compared according to
the collating sequence of the character set.   If the two
character strings have different lengths, the shorter string
value is extended on the right with character-fill characters
(blanks) before comparison.

The equality and inequality operations are also pre-defined for operand pairs of the following modes:

'POINTER'   (PT, PT)

'ENTRY NAME'   (EN, EN)

4.2.3 Boolean operations

The Boolean (logical) operations are defined on operands of Boolean and Bit modes only; they all produce Boolean value results which depend upon the values of the operands. Bit mode operands are converted to Boolean mode.

Logical negation (unary), denoted by "¬" and ".NOT."; e.g., "¬ P", ".NOT. P".

Logical "and" (conjunction) (binary), denoted by "&" and ".AND."; e.g., "P & Q", "P .AND. Q". If either operand is 'FALSE', the other operand expression possibly may not be evaluated.

Logical "or" (disjunction) (binary), denoted by "|" and ".OR."; e.g., "P | Q", "P .OR. Q". If either operand is 'TRUE', the other operand expression possibly may not be evaluated.

Logical "exclusive or" (binary), denoted by ".EXOR."; e.g., "P .EXOR. Q".

Logical "implication" (binary), denoted by ".THEN."; e.g., "P .THEN. Q". The result is 'FALSE' if the 1st operand is 'TRUE' and the 2nd operand is 'FALSE'; otherwise, the result is 'TRUE'. If the 1st operand is 'FALSE' or the 2nd operand is 'TRUE', the other operand expression possibly may not be evaluated.

Logical "equivalence" (binary), denoted by ".EQV."; e.g., "P .EQV. Q". The result is 'TRUE' if the operand values are equal, and 'FALSE' otherwise.

4.2.4 Bit-string operations

The bit-string operations are defined on operands of all
modes except 'TRANSFER POINT' and 'ENTRY POINT'. The result is
always a bit-string value, with the same mode and length as the
1st operand.

The bitwise logical operations:

The operand values are treated as bit strings. The binary
operations "and", "or", and "exclusive or" require equal-length
operands.

Bitwise negation (unary), denoted by ".N."; e.g., ".N. A". Each
    bit of the result is the negation (complement) of the
    corresponding bit of the operand.

Bitwise "and" (binary), denoted by ".A."; e.g., "A .A. B". Each
    bit of the result is the "and" (conjunction) of the two
    corresponding bits of the operands.

Bitwise "or" (binary), denoted by ".V."; e.g., "A .V. B".   Each
    bit of the result is the "or" (disjunction) of the two
    corresponding bits of the operands.

Bitwise "exclusive or" (binary), denoted by ".EV."; e.g.,
    "A .EV. B".   Each bit of the result is the "exclusive or"
    of the two corresponding bits of the operands.

The bitwise shift operations:

The first operand value is treated as a bit string.   The
second operand must have an arithmetic mode or 'BIT' mode; its
value is converted (if necessary) to an integer value, which
must be non-negative and is used as the shift count. The result
is a new value; neither operand is affected.

Bitwise-logical left shift and right shift (binary), denoted by
    ".LS."   and   ".RS.",   respectively; e.g., "A .LS. J",
    "A .RS. J". The 1st operand value is shifted left (or
    right) by the. number of bit positions specified by the
    shift count. If the shift count is negative the operation
    is undefined.   The bit string stays the same length; bits
    shifted off either end are lost, and vacated bit positions
    are filled with 0 bits.

Bitwise-arithmetic left shift and right shift (binary), denoted
    by ".LSA." and ".RSA.", respectively; e.g., "A .LSA. J",
    "A .RSA. J". The first operand value is treated as a
    binary representation of a signed integer. It is shifted
    left (or right) by the number of binary digits specified by
    the shift count.   If the shift count is negative the

operation is undefined. The binary integer stays the same
length; it is shifted so as to preserve its sign, and
effect multiplication (or division) by a power of two.
Digits shifted off either end are lost.

4.2.5 Character-string operations

Concatenation (binary), denoted by "||" and ".CONCAT."; e.g.,
   "A || B", "A .CONCAT. B". Both operands must be of
   character-string modes: 'CHARACTER' or 'VARYING CHARACTER'.
   The result is a value -- the 1st operand value concatenated
   with (followed by) the 2nd operand value. The length of
   the result is the sum of the (current) operand lengths.
   The result mode is 'CHARACTER' if both operands are of
   'CHARACTER' mode, and 'VARYING CHARACTER' otherwise.

4.2.6 Selection operations

Selection by component name (binary), denoted by "$"; e.g.,
   "A $ @NAME". The 1st operand must be a reference of a
   structured mode allowing named components ('COMPONENT
   STRUCTURE' or 'ALTERNATE'). The 2nd operand must be a
   component name which names some component of the 1st
   operand. The result is a reference of the named component;
   its mode, length, and other attributes are obtained from
   the subtype-decln part of the component declaration.

Selection by subscript value (n-ary), denoted by ".TAG." or
   implied by the syntactic context " expression ("; e.g.,
   "A .TAG. I", "A(I)", "A(I,J)", "(EXP)(K)". The 1st operand
   must be a reference of a structured mode allowing numbered
   components ('FIXED ARRAY', 'VARYING ARRAY', 'COMPONENT
   STRUCTURE', 'ALTERNATE'). The remaining operands must have
   values convertible to integers (arithmetic or 'BIT' modes),
   and are interpreted as an ordered set of subscript values.

      If the 1st operand has 'COMPONENT STRUCTURE' or
   'ALTERNATE' mode, there must be exactly one subscript. The
   integer subscript value must be at least 1 and not greater
   than the number of declared components. If the subscript
   expression is a constant (with possible sign), then the
   mode and other attributes of the result are obtained from
   the subtype-decln in the component declaration. If the
   subscript is not a constant, the attributes of the result
   cannot be synthesized by the compiler; then the attributes
   are considered "undeclared", and are usually attached with
   an .ASTYPE. or .ASTYPEOF. declaration.

      If the 1st operand has an array mode, there must be
   exactly as many subscripts as the array's dimension

attribute specifies. Each integer subscript value must be
in the range defined by the corresponding lower and upper
subscript bounds; otherwise the result is undefined. The
mode and other attributes of the result are obtained from
the subtype-decln in the mode declaration of the array.

In any case, the result is a reference of the selected
component.

Substring selection (ternary), denoted by ".TAG." or implied by
the syntactic context " expression ("; e.g., "CH(I)",
"A .TAG. (I,J)", "CH(I,J)". The 1st operand must have a
character-string mode ('CHARACTER' or 'VARYING CHARACTER'),
and may be either a reference or a value. Its value is the
character string (possibly null) in which a substring is to
be selected. Let $\underline{S}$ denote the string and let $\underline{m}$ be the
current string length. The 2nd and 3rd operands must have
values convertible to integers. Let $\underline{j}$ and $\underline{k}$ be the integer
values of the 2nd and 3rd operands, respectively; these are
interpreted as the position and length of the desired
substring. We require that $j>0$ and $k\geq0$. The 3rd operand
may be omitted; if it is, $k=1$ is assumed. The 3rd operand
may also be the special symbol # ; if it is, $k=m-j+1$ is
assumed. If $j>m$ or if $j+k-1 > m$, the operation is
undefined. Otherwise the substring is $S(j)$ -- $S(j+k-1)$.
The result is a reference or value according as the 1st
operand is a reference or value. If the 3rd operand
expression is an integer constant or omitted, then the
result is 'CHARACTER' mode with length k; otherwise the
result is 'VARYING CHARACTER' mode, with current length k.

4.2.7 Procedure-call operation

Procedure-call (n-ary) is denoted by "."; e.g., "F.X",
"G.(X,Y)". The 1st operand must have either 'ENTRY POINT' or
'ENTRY NAME' mode; it may be either a reference or value. This
operand identifies a procedure entry point to be called. The
remaining operands (if any) may be references or values; they
are the actual parameters to be passed to the procedure. Those
parameters which are values are held in temporary storage, and
are replaced by references of their allocated storage.

There are also two phrase keywords which may appear after
the actual parameter list; these are 'ERROR EXIT' and 'SAVE
CODE', and are used to examine a possible auxiliary "return
code" from the called procedure. 'ERROR EXIT' introduces a list
of labels; the labels denote places to "go to" for various non-
zero return code values. 'SAVE CODE' must be followed by an
'INTEGER LONG' variable; it is used to save the return code for
later reference.

Examples:

        RANDOM.
        F.(X)
        SORT.(N,VA,VB)
        GETLINE.(LINE 'ERROR EXIT' L1)
        FN.(P,Q 'ERROR EXIT' L1,L2 'SAVE CODE' RC)

The procedure-call proceeds as follows:

(1)   Evaluate the 1st operand expression to determine the
      desired entry point.

(2)   Evaluate the operand expressions for the actual parameters.
      Convert each 'ENTRY POINT' result to 'ENTRY NAME' mode, and
      assign the current environment information.   (The entry
      point named must be owned by the current block, but this
      cannot be checked by the compiler.)   Disallow 'TRANSFER
      POINT' mode. Allocate temporary storage for those operands
      which are values, and let the actual parameter list be
      references of the operands.

(3)   Save the current program position and environment
      information, and transfer control to the procedure entry
      point, in such a way that execution of a 'RETURN' will
      cause control to be resumed at (4) below.

(4)   If a 'SAVE CODE' phrase appears in the procedure call,
      assign the return code to the integer variable. (See
      Section 14 for implementation.)

(5)   If 'ERROR EXIT' appears in the procedure call, examine the
      return code. If the return code is zero, proceed to (6)
      below; otherwise, transfer control to the statement named
      by the k-th label if the return code is k, k=1,2,... If
      the return code exceeds the number of labels, the action is
      undefined.

(6)   The result is the value returned from the procedure; its
      attributes are obtained from the subtype-decln part of the
      1st operand declaration.

4.2.8   Conversion operations

MAD/I provides a number of operations to convert a value of
one mode to a corresponding value of another mode. In general,
the result is a new value, obtained by copying and transforming
the original value. Most conversions are implied by context and
automatically generated by   the   compiler.   However,   the
operations are all binary, and are denoted as a class by the
".CONV." operator; e.g., "A .CONV. ('INTEGER')".   This operator

requires a parenthesized decln-string as its 2nd operand expression.

The pre-defined conversions are described below. In the context table, each position represents a potential conversion from the row mode to the column mode. A "0" entry means that the conversion is defined and is trivial; other entries refer to the text following the table.

".CONV."	FL	FS	IL	IS	PK	BT	BL
FL	0	A	F	F	F		
FS	A	0	F	F	F		
IL	E	E	0	B	D		
IS	E	E	B	0	D		
PK	E	E	C	C	0		
BT			G	G		0	H
BL							0

(A)   The value is extended (or truncated) on the low-order end to the new length.

(B)   The value is extended (or trancated) on the high-order end to the new length. Truncation of a value not representable in the new mode will produce an erroneous result.

(C)   The value is converted from decimal to binary, and truncated (if necessary) to the new length. Information may be lost if the value is too large.

(D)   The value is converted from binary to decimal; the result is 'PACKED'(16).

(E)   The value is converted to binary (if necessary), then to un-normalized floating-long, then normalized, and finally truncated (if necessary) to the new length.

(F)   The value is extended (if necessary), to floating-long, then de-normalized to align the integral part, then converted to integer-long, and finally (if necessary) truncated or converted to decimal.

(G)   The bit-string value is interpreted as an unsigned binary integer, and extended with zeros (if necessary) on the high-order end to the new length.

(H)   The bit-string is interpreted as 'FALSE' if all bits are 0, and as 'TRUE' otherwise.

4.2.9 Assignment operations

Assignment of a value is a binary operation, denoted by
":="; e.g., "A := B", "VAR := 100". The 1st operand must be a
reference other than a label, and not of 'TRANSFER POINT' or
'ENTRY POINT' mode. The 2nd operand may be a reference or a
value.

The 1st operand expression is evaluated to produce a
reference. Then the 2nd operand expression is evaluated. The
value of the 2nd operand is converted (if necessary) to the mode
and storage-layout attributes of the 1st operand, and replaces
the value identified by the 1st operand. The result is a
reference of the 1st operand.

Assignment is pre-defined for the following contexts; some
notes are provided to fill in details which are not obvious.

    (Arithmetic mode, Arithmetic mode)

(BL, BL)	'BOOLEAN'
(BT, BT)	'BIT' -- extend/truncate on left.
(BT, BL)	Set all bits 1 ('TRUE') or 0 ('FALSE').
(BT, IL)	Express integer as bit string.
(BT, IS)	Express integer as bit string.
(C, C)	'CHARACTER'; extend/truncate on right.
(VC, VC)	'VARYING CHARACTER'
(VC, C)	Set current length = fixed length.
(C, VC)	Extend/truncate on right.
(PT, PT)	'POINTER'
(EN, EN)	'ENTRY NAMF'

('ENTRY NAME', 'ENTRY POINT') The entry name value points
    to the entry point, and the current environment
    information is assigned. The entry point must be
    owned by the current block; this cannot (in general)
    be checked by the compiler.

('ENTRY NAME', 'TRANSFER POINT') The entry name value
    points to the transfer point; the environment
    information is undefined. The resulting value can be
    used in a 'GO TO', but not in a procedure call, nor as
    an actual parameter.

4.2.10 Other operations

Length of a value (unary), denoted by ".LN."; e.g., ".LN. B".
  The operand may be a reference or value of any mode other
  than 'TRANSFER POINT' or 'ENTRY NAME'. The result is a
  value of 'INTEGER LONG' mode -- the "length" of the operand
  value. For 'VARYING CHARACTER' operands, the "length" is
  the current length of the character string.

Association of storage (binary), denoted by ".ALLOC."; e.g.,
  "A .ALLOC. B". The 1st operand expression must be a
  variable of based or formal parameter storage class, and of
  any mode. The 2nd operand must be a reference or value of
  'POINTER' mode. The storage reference determined by the
  pointer value is associated with the variable, so that the
  variable now has this reference as its result. If the
  pointer value equals 'NULL PT', then the variable becomes
  "not allocated", and its result is undefined.

Create pointer (unary), denoted by ".PT."; e.g., ".PT. B".   The
  operand must be a reference (of any mode). The result is a
  value of 'POINTER' mode corresponding to the reference;
  i.e., a pointer to the operand.

Indirect reference (unary), denoted by ".IND."; e.g., ".IND. B".
  The operand must have 'POINTER' mode; its value must be a
  non-null pointer. The result is the reference determined
  by the pointer; the mode and storage layout attributes are
  obtained from the subtype-decln part of the pointer
  declaration.

Create a pointer constant (unary; compile-time only), denoted by
  ".PTCON."; e.g., ".PTCON.(B)". The operand expression must
  be an identifier, and must be enclosed in parentheses. The
  result is a constant of 'POINTER' mode corresponding to the
  reference of the identifier.

Create an entry-name constant (unary; compile-time only),
  denoted by ".ENCON."; e.g., ".ENCON.(B)". The operand
  expression must be an identifier, and must be enclosed in
  parentheses. The result is a constant of 'ENTRY NAME'
  mode; it points to the entry point named by the identifier,
  but it does not carry environment information.

4.3 Operator Precedence and Class

MAD/I operators are symbols which denote operations (see Sec. 2.2.4). The operations themselves are described in the preceding subsection; we now describe the syntactic properties of operators.

Every operator has a <u>syntactic class</u> and a <u>precedence level.</u> The syntactic class tells how the operator is written with respect to its operand expressions:

Prefix: <u>before</u> its operand expression(s).

Postfix: <u>after</u> its operand expression(s).

Infix-left: <u>between</u> its operand expressions; associates left-to-right with operators of equal precedence.

Infix-right: <u>between</u> its operand expressions; associates right-to-left with equal-precedence operators.

An operator's precedence level (precedence) determines its syntactic "binding strength" relative to other operators. An expression appearing between two operators is "bound" as an operand expression to one operator or the other as follows:

If the operators have different precedence levels, the expression is bound to the higher-level operator.

If the operators have the same precedence level, they must be either both infix-left or both infix-right. The expression is bound to the left operator if they are infix-left, and to the right operator if they are infix-right.

To avoid the possibility of ambiguous constructions, a rule is applied to all operators, both pre-defined and user-defined:

All operators having the same precedence level must have the same syntactic class.

Also, parentheses may be used as grouping marks in the usual way: one or more expressions (separated by commas) may be enclosed in parentheses, forming a "group" of expressions which is bound as a unit. This is often necessary in denoting n-ary operations; e.g., ARRAY .TAG. (I,J,K) .

The following table shows the pre-defined operators, arranged from highest precedence level to lowest precedence level, and the syntactic class at each level. (There are no pre-defined postfix operators.)

INFIX LEFT	INFIX RIGHT	PREFIX	OPERATORS
X			.TAG. . @ .CONV. .ASTYPE. .ASTYPEOF.
		X	.ABS. .LN. .PT. .IND. .PTCON. .ENCON.
X			.LS. .RS. .LSA. .RSA.
		X	.N.
X			.A.
X			.V. .EV.
	X		**
		X	.NEG.
X			* / .REM.
X			+ -
X			|| .CONCAT.
X			= ¬= .NE. > >= < <=
		X	¬ .NOT.
X			& .AND.
X			| .OR. .EXOR.
X			.THEN.
X			.EQV.
	X		:= .ALLOC.

Part I -- Description of the MAD/I Language

4.4 Syntax of Expressions

The set of MAD/I operators is extensible, and new operators
may introduce new precedence levels between the existing levels.
Therefore we must resort to unconventional methods to present a
syntax which will describe all possible expressions.

Let precedence levels be denoted by special variables: i,
j, k, l. Let notations such as ">j" mean "any level higher than
level j". Also let the notation "i°j" mean "the lower of levels
i and j", and let "+" denote the highest possible level.

Associate with syntax variable "exp" (for "expression") two
precedence level parameters: the precedence "viewed from the
left", and the precedence "viewed from the right".   Thus, the
syntax notation "exp(i,j)" will denote an occurrence of an "exp"
with precedence levels i and j as viewed from the left and
right, respectively.

Also define syntax variables for the operators, with their
precedence levels as parameters. Thus, "prefix-op(j)" denotes
an occurrence of a prefix operator with precedence level j, and
similarly for postfix-op(j), infix-L-op(j), and infix-R-op(j).

In this extended notation, we now define the formal syntax
of MAD/I expressions. An example rule is explained below.

exp(+,+)   = constant | identifier | embedded-statement
                | (list , exp)

exp(+,j°k)   = prefix-op(j) exp(>j,k)

exp(i°j,+)   = exp(i,>j) postfix-op(j)

exp(i°j,j°k)   = exp(i,≥j) infix-L-op(j) exp(>j,k)

exp(i°j,j°k)   = exp(i,>j) infix-R-op(j) exp(≥j,k)

expression   = exp

For example, the syntax rule

exp(+,j°k)   = prefix-op(j) exp(>j,k)

means: "A prefix operator with precedence level j, followed by
an expression with any left-precedence greater than j and any
right-precedence k, forms a composite expression with left-
precedence "highest" and right-precedence equal to the lower of
j and k". Referring to the operator table in Section 4.3, we
see that this rule can combine ".NEG." and "A**B" to get
".NEG.A**B", but it cannot combine ".NEG." and "A+B" to get
".NEG.A+B".

Section 5: Statements

5.0 Introduction

Each non-empty statement in the language falls into one of
five classes: (i) simple statements, (ii) compound statements,
(iii) prefix statements, (iv) list statements, and (v)
declaration statements. Unless otherwise indicated by the
interpretation of a statement, its successor (at run time) is
the statement written immediately after it. Two adjacent
statements are always separated by a semicolon, but the
semicolon is not a part of the statement it follows.

Empty statements

A statement can also be "empty" (consisting of no symbols).
An empty statement specifies no computation; it can, however, be
labeled.

Syntax: statement = empty

Labeled statements

Any statement can be labeled, by prefixing it with an
identifier and a colon; the resulting form is itself a
statement. Labels on declaration statements are permitted.

Syntax: statement = identifier : statement

Simple statements

The simple statements have two general forms:

(1) a single expression;

(2) a simple-statement keyword, possibly followed by one or more
    expressions separated by commas or phrase keywords (a
    "phrase list").

In case (1), the expression is simply evaluated; it has no
effect other than the effects produced under the rules of
expressions; the result is not saved.   Such a statement is
usually an assignment, an .ALLOC. expression, or a procedure
call.

In case (2), the exact statement form is determined by the
statement keyword and its associated statement definition. For

Part 1 -- Description of the MAD/I Language

each statement keyword, there is a fixed number of expressions
which may follow.

Syntax: statement = expression
                      | simple-stmt-keyword [phrase-list]

            phrase-list = <u>list</u> { , | phrase-keyword } expression

Examples: A := B
           FN.(X,Y)
           'GO TO' LB

Compound statements

A compound statement is simply a sequence of statements
separated by semicolons and bracketed by a compound-statement
keyword and an end keyword. The resulting form is itself a
statement.

Syntax:
        statement = compound-stmt-keyword stmt-seq end-keyword

        stmt-seq = <u>list</u> ; statement

Example:
        'BEGIN' A := B; B := C 'END'

Prefix statements

Each prefix statement form begins with a "prefix part",
consisting of a prefix-statement keyword and a fixed-length
phrase list, such as:

        'IF' exprn
or
        'FOR' desig := exprn, exprn, exprn

For each such prefix part there are two forms of the prefix
statement: (1) the prefix-part followed by a comma and a single
statement (the "short form"); (2) the prefix-part, possibly
followed by a semicolon and a statement-sequence, and ending
with a matching end keyword (the "long form").   The particular
end keyword which "matches" depends upon the statement keyword;
however, the symbol 'END' is a general-purpose end keyword which
may be used to end any long-form prefix statement.

Prefix statements and compound statements may be properly
nested; each occurrence of a long-form prefix statement requires
its own end keyword.

Part I -- Description of the MAD/I Language

Syntax:

```
statement = prefix-stmt-keyword phrase-list
            { , statement
            | [; stmt-seq] end-keyword }
```

Short-form example:

```
'IF' A > B, B := A
```

Long-form examples:

```
(1)      'FOR' I := 1, 1, I > N;
          V(I) := I + 1; W(I) := 0 'ENDFOR'

(2)      'FOR' J := 0, 1, C(J) = 0 'ENDFOR'
```

Note that in example (2), the prefix is followed immediately by
the end keyword; the "scope" of the statement is empty.

List statements

A list statement consists of a prefix followed by a varying
number of expressions. The prefix begins with a statement
keyword; the form of the rest of the prefix depends upon the
particular statement keyword.

Examples:

```
'WRITE' ("3I7*",1), J, K, L

'PRESET' A:=3, V(1):=1, V(3):=3

'LIST' X(I), Y(I), Z(I)
```


Declaration statements

Declaration statements have two general forms: the
'DECLARE' statement and the "inverted" declaration statements.
They have a special syntax, described in Sections 3.6 and 5.9.

Examples:

```
'DECLARE' A 'INTEGER', (B,C,D) 'FLOATING'

'BOOLEAN' S1, S2, S3
```

5.1 Expression Statements

An expression is also a simple statement. Execution of an
"expression statement" consists of evaluating the expression and
ignoring the result.

Notice that expressions include assignments and procedure
calls.

```
Examples:   V := A + B
            BV .ALLOC. (.PT. V)
            SORT. (N, A1, A2, KEY)
            A + B
```

5.2 The 'GO TO' Statement

This (simple) statement has the form:

'GO TO' S

Here S may be any label or entry point or any expression in entry name mode. Execution of this statement causes the computation to continue at the statement whose label is the value of S.

Examples:

'GO TO' LOOP4

'GO TO' ENTRYB

If the value of S is an entry point (i.e., if it has appeared in a 'PROCEDURE' definition or has been declared 'ENTRY POINT') a 'GO TO' statement may be used, even if the statement it labels is not in the same program. In addition, for entry points one can get parameter substitutions at the same time by a 'WITH' clause containing a parenthesized list of actual parameters:

'GO TO' S 'WITH' (E(1), E(2), ―, E(N))

where the expressions E(i) agree in mode and length with the formal parameters declared for the entry point designated by S.

5.3   The 'IF' Statement

This (prefix) statement has a prefix of the form

'IF' bool-exprn

where "bool-exprn" is an expression of Boolean mode. Thus, examples of the short form are:

'IF' X > Y, 'GO TO' S1

'IF' A = B & I = J, Q := R + .ABS. T

The general long form is:

'IF' bool-exprn ; stat-seq
        [list {; 'OR IF' bool-exprn ; stat-seq}]
        [; 'OR ELSE'; stat-seq] 'ENDIF'

stat-seq = list ; statement

The 'OR IF' groups are optional; any number of them may be used.   The 'OR ELSE' group is likewise optional, but only one 'OR ELSE' may appear in a given long-form 'IF' statement.   'OR ELSE' may be abbreviated as 'ELSE'.

The effect of this statement is to select for execution one of the statement sequences "stat-seg". Specifically, the first Boolean expression "bool-exprn" from the left found to be true causes the execution of the immediately following "stat-seq". Here, 'OR ELSE' can be interpreted as "always true", i.e., as 'OR IF' 'TRUE'.

Example long-form 'IF' statements:

'IF' A = B; S :=T + J; I := I-1; 'GO TO' M 'END IF'

'IF' L < S; I := I+1; 'OR IF' Q > S; I := I-1;
        'OR ELSE'; 'GO TO' ST 'ENDIF'

'IF' J=0; D(J):=1; 'ELSE'; D(J):=D(J)+1 'ENDIF'

5.4 The 'FOR' Statement

The 'FOR' statement is a prefix statement for specifying
iterations. The statement-keyword and phrase-list are:

'FOR' desig := exprn2, exprn3, exprn4

where "desig" produces a reference of the iteration value,
"exprn2" gives the initial value, "exprn3" gives the increment
value, and "exprn4" is a Boolean expression to test for
completion. The modes of exprn2 and exprn3 must be such that
"desig := exprn2" and "desig := desig + exprn3" are legitimate
expressions. The end-keyword for the 'FOR' statement is
'ENDFOR'.

The interpretation of the 'FOR' statement with scope "stmt-
seq" is as if it had been written as follows:

```
      desig := (exprn2);
L:    'IF' ¬ (exprn4);
      stmt-seq;
      desig := (desig) + (exprn3);
      'GO TO' L
      'ENDIF'
```

where "L" represents a local label. In other words: the
designator is evaluated to get the reference for the iteration
value; the iteration value is initialized to the value of
exprn2; as long as the value of exprn4 is 'FALSE', the scope
stmt-seq is executed, followed by incrementing the iteration
value by the latest value of exprn3. Note that if exprn4 is
'TRUE' on the first test, the scope is not executed at all.

Examples:

```
(1)        'INTEGER' J,N;
           SUM := 0.;
           'FOR' J := N, -1, J < 0, Y := SUM * X + C(J)

(2)        'FOR' I := 1, 1, CH(I) = "," | I > K, ;
           'IF' I > K, 'GO TO' NOCOMMA

(3)        'FOR' I := 1, 1, I > M;
             J := 0;
             'FOR' S(I) := 0, B(I,J), (J := J + 1) > M
             'ENDFOR'
           'ENDFOR'
```

5.5   The 'FOR VALUES' Statement

The 'FOR VALUES' statement is another prefix statement for specifying iterations. The prefix has the form:

'FOR VALUES' desig := (list , exprn)

where "desig" designates the iteration value, and each "exprn" in the list has a mode such that "desig := exprn" is a valid assignment. The end keyword for the 'FOR VALUES' statement is 'ENDFOR'.

The interpretation of the 'FOR VALUES' statement is as follows:

(1)   Evaluate "desig" to determine the iteration value.

(2)   Set (local variable) i equal to 1.

(3)   Evaluate the i-th "exprn" in the list, and let its value (with the appropriate conversion, if necessary) replace the iteration value.

(4)   Execute the scope (statement or statement sequence). Let normal sequencing proceed to (5).

(5)   If i is equal to the number of "exprn"s in the list, the 'FOR VALUES' statement is finished; otherwise, increment i by 1 and go back to (3).

Examples:

(1)   'FOR VALUES' K := (0,1,5), A(K) := 0

(2)   'FOR VALUES' CH := ("A", "X", "0", "1");
             J := SCAN.(LINE, CH);
             'IF' CH = "X", JX := J
            'ENDFOR'

5.6 The 'VALUE' Statement

This (prefix) statement has a prefix of the form

'VALUE' V := E

where V is a designator, and E is an expression such that the assignment V := E is legitimate. A shorter form of the prefix is

'VALUE' V

in which case the initial value of V is the value it had just before execution of the 'VALUE' statement. The end keyword is 'END VALUE'.

An example of the short form is:

'VALUE' S := 0., 'FOR' I := 1,1,I > N,

S := S + A(I)

An example of the long form is:

'VALUE' TRACE := 0.;

'FOR' I := 1,1,I > N;

'FOR' J := 1,1,J > N;

C(I,J) := 0.;

'FOR' K := 1,1,K > N,

C(I,J) := C(I,J) + A(I,K)*B(K,J)'END FOR' ;

TRACE := TRACE + C(I,I)'END FOR' 'END VALUE'

The interpretation of the 'VALUE' statement is that a value is produced for V as a result of the execution of the scope. This prefix statement may now be enclosed in parentheses and used as an embedded statement, since it has produced a value. The expression E in the prefix is an initial value for V. Thus, in the long-form example above, if N = 0, then none of the scope would actually be executed (since I > N), and the value produced (which in any case is the final value of TRACE), is 0.

5.7 Procedures

5.7.1 The 'PROCEDURE' Statement

This (prefix) statement, called a procedure definition, has
the following syntax:

    procedure = proc-prefix; list ; statement 'END PROCEDURE'

    proc-prefix = 'PROCEDURE' list , entry-spec

    entry-spec = identifier-list [.][formal-parameters]

    formal-parameters = (list , identifier)

A typical prefix would be:

        'PROCEDURE' (J,K,L).(X,Y,Z,W)

where the first part specifies entry points for the procedure,
i.e., J, K, and L, and the second part specifies formal
parameters to be associated with each of those entry points. If
there is only one entry point, the parentheses around it may be
omitted. If there are no formal parameters, the second pair of
parentheses may be omitted. The period is always optional in a
procedure prefix. Thus, the prefix

        'PROCEDURE' (F,G).(X), H.(X,Y), L.

specifies that F and G are entry points with formal parameter X,
that H is an entry point with parameters X and Y, and that L is
an entry point with no parameters.

The short form of the 'PROCEDURE' statement differs
somewhat from the usual short form; it looks much like an
assignment expression:

        procedure-short =
                'PROCEDURE' identifier [.] formal-parameters
                        := expression

where "expression" is any expression (possibly an embedded
statement). As an example we have:

        'PROCEDURE' REM.(A,B) := A - (A/B)*B

The long form uses the usual sequence of statements,
separated by semicolons and terminated by the end keyword
'END PROCEDURE'. Each entry point occurring in the   prefix may
appear as a label on some statement in the scope of the
'PROCEDURE' prefix. If no such label appears on any statement,

it is as if the label were on the first executable statement
within the definition of the procedure. Procedure definitions
may be properly nested within other procedure definitions.

Procedures are _defined_ at compile time only; at run time, a
procedure statement in a statement sequence behaves as an empty
statement in that sequence.

5.7.2 Formal Parameters

The formal parameters of a procedure are local variables
which are dynamically "bound" to their storage references when
the procedure is entered. All formal parameters declared in the
procedure prefix are variables usable throughout the procedure
body. For each formal parameter, however, only certain entry
points cause it to be bound -- namely those entry points whose
"entry-spec"s mention that formal parameter.

A formal parameter, like any other variable, acquires mode
and storage-layout attributes. These may be declared (within
the procedure) in any of the ways described in Section 3.

Whenever (at run time) a procedure is entered at a given
entry point, the formal parameters specified for that entry
point are considered in the order declared and bound to the
actual parameters (arguments) received from the calling
procedure. There must be at least as many actual parameters as
formal parameters; each actual parameter must be a reference of
the _same mode_ as the corresponding formal parameter. Generally
the storage-layout attributes must also agree, but there are a
few permissible exceptions:

'CHARACTER' mode: The length of the actual parameter may be
     greater than the length of the formal parameter.

'VARYING CHARACTER' mode: The maximum length of the actual
     parameter may be greater than that of the formal parameter.

Array modes: The formal parameter may optionally be declared
     with an "array-suffix" in which all the "bounds" entries
     are the special symbol # . In this case, the number of
     dimensions of the actual and formal parameters must agree,
     but the bounds values and storage spacing of the formal
     parameter are taken from those of the actual parameter (cf.
     Section 3.1.2.1). For example:

          'DCL' AA 'FIXED ARRAY' (#,#) 'FLOATING'

However, for 'FIXED ARRAY' parameters, greater efficiency
can often be realized if the actual bounds are known and
declared in the procedure.

Part I -- Description of the MAD/I Language

5.7.3   Procedure Returns

The execution of a procedure ends when any of 'RETURN',
'RETURN TO', or 'END PROCEDURE' is executed. The forms of these
statements are:

(i)   'RETURN' [expression] [,return-code]

where return-code is the return code value and the expression is
the result value of the procedure. If the return-code is
missing, a return-code of zero is given. If a return-code is
given, it must be a non-negative integer expression. The
"return" is made to the point immediately after the last "call"
was executed.

(ii)   'RETURN TO'   S

where S is (1) a formal parameter of the current procedure, and
(2) has 'ENTRY NAME' mode and has an actual parameter value
which is an entry name for some procedure whose call preceded
the current one in the currently effective chain of "calls".
For example, suppose procedure A1 has called A2, which has
called A3, each call passing as a parameter the entry name L in
A1. Then A3 might contain the statement:

'RETURN TO' S   · ·· ⁻

where S is a formal parameter for which L is the actual
parameter. The next statement executed after the 'RETURN TO'
statement is that denoted by the value of L.

S also can be a variable of 'ENTRY NAME' mode which has
been assigned a value by means of an assignment operation
located in the procedure which owns the associated entry point.
At the time the 'RETURN TO' is executed, the entry name variable
must have a value which points to a currently active block; that
is, the environment information must still be valid.

The execution of 'END PROCEDURE' , which ends the scope of
a procedure, is permissible and is equivalent to the execution
of 'RETURN' with no result value and no return-code specified.

5.8   Input/Output Statements

There are several statements for specifying input/output operations; they are mentioned below. For a complete treatment of input/output, refer to Section 6; the statements are described in Sections 6.8 and 6.9.

```
'OPEN'
'CLOSE'
'READ DATA'
'WRITE DATA'
'READ'
'WRITE'
'READ UNCONVERTED'
'WRITE UNCONVERTED'
```

5.9 Declaration Statements

The statements in this section have a purely "compile-time" effect; at run time, they act as "empty" statements.

The 'DECLARE' statement and the inverted declaration statements are described in Section 3.6; refer to that section.

The 'DECLARE' statement (abbreviation 'DCL') -- Section 3.6.1.

Inverted declaration statements -- Section 3.6.2.

The 'DECLARE DEFAULT' statement (abbreviation 'DCLD') is used to declare default mode and storage class attributes. It can also be used to control the scope of identifiers referenced but not declared.

Syntax: 'DECLARE DEFAULT' decln-string

This statement has the same effect as the statement

        'DECLARE' 'DEFAULT' decln-string

which is described in Section 3.7.2.

Example: 'DECLARE DEFAULT' 'INTEGER'

5.10   The 'BEGIN' and 'BLOCK' Statements

The 'BEGIN' and 'BLOCK' statements are compound statements, consisting of a sequence of constituent statements bracketed by the compound-statement keyword and an end keyword.

Syntax: statement = 'BEGIN' stmt-seq 'END'
                      | 'BLOCK' stmt-seq 'END'

The 'BEGIN' statement serves only to treat the statement sequence as a single statement -- it has no other effect. Execution of a 'BEGIN' statement means execution of the statement sequence.

The 'BLOCK' statement is the same as the 'BEGIN' statement, except that it forms a new block (see Section 7). The scopes of names and declarations appearing within the 'BLOCK' statement are determined with respect to this block.

Examples:
        'BEGIN' T := A; A := B; B := T 'END'

        'BLOCK'
          'EXTERNAL' (B1,B2,B3) 'BOOLEAN';
          ANYB := B1 | B2 | B3
        'END'

5.11 The 'PRESET' Statement

'PRESET' is a statement used to specify initial values of
variables. An "initial value" is a value assigned to a variable
at the time storage is allocated for the variable. Storage for
'STATIC' variables is considered to be determined at compile
time and allocated just prior to run time. Only 'STATIC'
variables which are not 'EXTERNAL' may be preset.

<u>Syntax:</u>

       statement = 'PRESET' <u>list</u> , pre-assign

       pre-assign = pre-var := { <u>list</u> , init-value }

       pre-var = variable [<u>list</u> { (<u>list</u> , integer) }]

       init-value = const-exprn
                   | replic-exprn
                   | empty

       const-exprn = constant
                   | - constant-exprn

       replic-exprn = unsigned-integer (<u>list</u> , init-value)

<u>Examples:</u>

of pre-var: BCX
                   AA(1,-3,2)
                   CC(2,1)(4)(17)

of const-exprn: 20
                   "Haw!"
                   -4 @ ('INTEGER SHORT')
                   .ENCON. LOGTAN

of replic-exprn: 3(1.2, 7, "AB")
                   300(0)
                   20(1.0,8(0.0),2.0)

of preset statements:

       'PRESET' A := 0, B := 0, CH := "0123"

       'PRESET' V(1) := 2., V(4) := 0., V(10) := 10.

       'PRESET' AB(1,1) := 1, 1, 2, 0, AB(2,1) := 4(0)

Interpretation

A pre-var specifies either a variable to be preset or a
component of a variable at which presetting is to start. The

Part I -- Description of the MAD/I Language

list of init-value's following the ":=" specifies a sequence of
constant values to be pre-assigned to the pre-var. If the pre-
var is a variable of a primitive mode, there should be only one
init-value in the list. If the pre-var is an array or
component-structure variable, then presetting begins with the
1st component and continues with successive components at the
same structural level. If the pre-var is a component of such a
variable, presetting begins with that component and continues as
above.

An empty init-value causes the corresponding component to
be skipped without being preset. A "replic-exprn" is treated as
an abbreviation for the enclosed list of init-value's written
out "unsigned-integer" times. The unsigned-integer must be non-
zero. For example,

        2(1,2, ,4)

is equivalent to

        1,2, ,4,1,2, ,4 .

The initial values must have the same mode as their
corresponding variables or components; no automatic conversion
is performed.

5.12 The 'DECLARE CSECT' and 'DECLARE PSECT' Statements

'DECLARE CSECT' and 'DECLARE PSECT' are both simple
statements used to control the names given by the compiler to
sections of the object module.

Syntax: statement = 'DECLARE CSECT' identifier
                    | 'DECLARE PSECT' identifier

The compiler normally produces an object program segregated
into two sections: (1) a section ("csect") which is never
modified as the program is run and is "shareable" by different
recursion levels in the task and by different tasks in the
operating system, and (2) a section ("psect") which contains all
the rest -- the variable values and non-shareable text. The
programmer may occasionally need to specify the names given to
these sections.

These statements cause the specified identifier to be used
as the name for the specified section. It is the programmer's
responsibility to make sure that the name is acceptable to the
operating system in which the object program will be run.

5.13 The 'ALLOCATE' and 'DEALLOCATE' Statements

These are simple statements which dynamically allocate and de-allocate storage for variables of based storage class.

Syntax: statement = 'ALLOCATE' variable [, exprn]
                    | 'DEALLOCATE' variable

The 'ALLOCATE' statement specifies a based variable to receive a new allocation. The "exprn", if included, must be integer-valued, and specifies the number of contiguous storage locations (bytes) to be allocated; if the expression is omitted, the length attribute (Sec. 3.2.1) of the variable is used. The storage is acquired (from the operating system) and associated with the based variable. If storage was already allocated for that variable, the variable's reference of that storage is lost (i.e., not saved or automatically freed).

The 'DEALLOCATE' statement is used only to de-allocate the storage allocated to a based variable by an 'ALLOCATE' statement. The specified variable is set to "not allocated", and the storage previously allocated for it is freed (returned to the operating system). If the variable has storage which was allocated by means other than the 'ALLOCATE' statement, then the action of 'DEALLOCATE' is undefined.

Examples:

        'ALLOCATE' BLOCK

        'DEALLOCATE' BLOCK

        'ALLOCATE' MATRIX, M*N*4

        'DEALLOCATE' MATRIX

5.14   The 'REDIMENSION' Statement

'REDIMENSION' is a statement for dynamically modifying the dimension attributes of 'VARYING ARRAY's at run time. Refer to Section 3.1.2.1 (Array modes) for information on 'VARYING ARRAY' mode.

Syntax: statement = 'REDIMENSION' list , TO-phrase

          TO-phrase = desig 'TO' (list , run-bounds)

          run-bounds = [exprn ...] exprn

Example:
          'REDIMENSION' AA 'TO' (M,0...N)

In the syntax above, "desig" denotes a designator, which must have 'VARYING ARRAY' mode. In each TO-phrase, the number of "run-bounds"s must equal the declared number of dimensions of the array designated by "desig". Also, "exprn" denotes any expression whose value is convertible to an integer. The (optional) 1st exprn specifies the lower bound for that subscript position; if omitted, a lower bound of 1 is assumed. The 2nd exprn specifies the corresponding upper bound.

For each TO-phrase, the run-bounds expressions are evaluated, and their values are converted (if necessary) to integers. The dimension attribute of the array denoted by the designator is changed to reflect the new subscript bounds. However, the storage allocated to the array is not changed; therefore, if the array is in an allocated state (i.e., storage is allocated for it), then the storage requirement of the re-dimensioned array must not exceed the amount of storage allocated.

For example, if we had declared:

          'DECLARE' (V1, V2) 'VARIABLE ARRAY' (100),
                AD 'VARIABLE ARRAY' (30,20)

then we might write re-dimension statements like these:

          'REDIMENSION' V1 'TO' (M)

          'REDIMENSION' V2 'TO' (0 ... N-1),
                AD 'TO' (0...K, L+1)

Section 6: Input/Output

Before discussing the input/output statements in detail, several general concepts should be defined.

6.1 Data sets, Records, and Files

A data set is a collection of data external to the program. Input activity transmits data from a data set to a program. Output activity transmits data from a program to a data set. A data set consists of discrete records, each consisting of zero or more bytes. An input activity, then, transmits one or more whole records from a data set to a program while an output activity transmits one or more whole records from a program to a data set. An input activity is also referred to as reading while an output activity is also referred to as writing.

A file is a usage of a data set. A file can be opened either explicitly or implicitly. A file is opened explicitly by means of an 'OPEN' statement. In this case the file is characterized by the value of the variable of 'FILE NAME' mode specified as a part of the 'OPEN' statement. Every explicitly opened file is a unique file even if it uses the same data set as another file. A file is opened implicitly through the execution of an input/output statement (other than 'OPEN') which references it with no prior implicit opening of the file. The file referenced is deduced from the data set name given in the input/output statement and the manner in which the data set name is specified. An implicitly opened file is characterized by the value of a variable of 'FILE NAME' mode owned by the system input/output support software. This filename variable cannot be referenced by name, but only implicitly through the specification of the same data set name in the same manner as when it was implicitly opened. This will be clarified in Section 6.3. Note that several files may be open which use the same data set. The behavior in this case is dependent upon the system and the type of data set organization.

6.2 Types of Input/Output Activities

There are four types of input/output activities supported
in MAD/I: data-directed, list-directed, format-directed, and
unconverted. This section describes the general characteristics
of these transmission modes.

6.2.1 Data-directed Transmission

Data-directed transmission permits the user to read or
write self-defining data.

Input: The data are in a form similar to a 'PRESET' statement,
consisting of a list of designators, each followed by an
assignment symbol (or equality symbol) and a list of constant
values to be assigned.   The input for a single data-directed
input transmission is free-form and may span one or more whole
records.   The transmission is terminated by a semi-colon in the
last input record. A typical input record is:

    A:=-3.2, B:="S", COMPLXN$aR:=1.5, Z(2):=1,,5(2) ;

Output: The data values to be transmitted are specified by a
data-list in the output statement. The data are placed into one
or more output records and consist of a list of designators,
each   followed   by the value referenced.   If a data-list
expression is not a designator (e.g., X+3), then three asterisks
(***) are printed in place of the designator. The records
produced by a data-directed output transmission are suitable as
input records for a data-directed input transmission; the items
identified with three asterisks are ignored.

6.2.2 List-directed Transmission

List-directed transmission permits the user to specify the
designators to which data are assigned or from which data are
transmitted, without specifying the format.

Input: The data are in the form of free-form constant values
separated by blanks or commas. The designators to which the
data are to be assigned are specified by a data-list in the
input statement.

Output: The data values to be transmitted are specified by a
data-list in the output statement. Each data item is converted

Part I -- Description of the MAD/I Language

to an external form (according to its mode and value), and the
external forms are concatenated to form output records.

6.2.3 Format-directed Transmission

Format-directed transmission permits the user to specify:
(1) the designators to which data are to be assigned or from
which data are to be transmitted, through a data-list, and (2)
the form of the data fields in the records, through a format
specification.

Input: The form of the data in the input records is defined by a
format specification. The designators to which the data are to
be assigned are specified by a data-list.

Output: The data values to be transmitted are defined by a data-
list.   The form that the data are to have in the output records
is defined by a format specification.

6.2.4 Unconverted Transmission

Unconverted transmission permits the user to read or write
information directly, with no conversion. The unconverted
input/output statements cause a single record to be transmitted
from or to the data set. The designators to which the data are
to be assigned or from which data are to be transmitted are
specified through a data-list.

6.3   Associating Data sets with Files

A data set is associated with a file at the time the file
is opened. The data set name can be specified in five different
ways in either the 'OPEN' statement (for explicitly opened
files) or an input/output statement other than 'OPEN' (for
implicitly opened files.) These five ways are: (i) through a
unit specification, (ii) through a data set name specification,
(iii) through a character-string specification, (iv) through an
entry-name specification, and (v) through a default
specification. Only one of these five ways can be used in any
one statement.

6.3.1   Unit Specification

A unit is a name which is associated with a particular data
set through the job control language of the operating system in
which the MAD/I program is being run. The unit is specified
through the 'UNIT' specification in the input/output statement.
This specification can be an arithmetic expression or a
character-string expression. The values of these expressions
are interpreted in a system-dependent fashion.

In input/output statements other than 'OPEN', the unit
specification can also be an expression of 'FILE NAME' mode, in
which case the named file is used. It must have previously been
opened in an 'OPEN' statement.

All implicit references to files which satisfy the
following two rules will be considered as references to the same
file:

1.   All references are by means of a unit specification.
2.   Either all references are by means of arithmetic
      expressions which compare as equal in value or all
      references   are   by   means   of   character-string
      expressions which compare as equal in value.

In MTS, the value of a character-string expression must be
the name of a "logical I/O unit". The valid logical I/O units
are: SCARDS, SPRINT, SPUNCH, GUSER, SERCOM, and the numbers 0
through 9.   The value of an arithmetic expression must be
integer-valued from 0 through 9 or the address of a "FDUB" as
returned by the subroutine GETFD. Non-integer values will be
truncated to the next lower integer value.

In OS, the value of a character-string expression must be a
current "ddname".   These names are defined through DD job
control language statements.   The value of an arithmetic
expression must be integer-valued from 0 through 99. Non-
integer values will be truncated to the next lower integer

value.

Examples: (using MTS conventions)

        'OPEN'('UNIT' "0",'END OF FILE' MACEND),MACLIB
        'OPEN'('UNIT' 0, 'END OF FILE' MACEND),MACLIB
        'OPEN'(0, MACEND),MACLIB

are all equivalent and open the file MACLIB which uses the data
set associated with the logical I/O unit 0.

        'READ DATA'('UNIT' MACLIB)
        'READ DATA'(MACLIB)

are equivalent and use the file MACLIB. If MACLIB were opened
with one of the above 'OPEN' statements, the data set ultimately
used would be the one associated with the logical I/O unit 0.

        'READ DATA'('UNIT' "SCARDS")
        'READ DATA'("SCARDS")

are both equivalent and   perform   a   data-directed   input
transmission using the data set associated with the logical I/O
unit SCARDS. The file associated with   the   unit   specification
"SCARDS" will be implicitly opened the first time it is
referenced through the execution of an input/output statement
which specifies it.

6.3.2 Data set Name Specification

The name of a data set can be specified through the 'DATA SET' specification in the input/output statement. This specification is done through a character-string expression. The value of this expression is interpreted in a system-dependent fashion.

All implicit references to files which satisfy the following two rules will be considered as references to the same file:

1. All references are by means of a data set name specification.
2. All references are by means of character-string expressions which compare as equal in value.

In MTS, the value of the character-string expression must be a file or device name ("FDname"). The name may be a concatenation of file or device names, each followed by modifiers or a line number range, as described in the MTS manual. The FDname need not be followed by a blank. Note that the MTS term "file" represents a different concept than the MAD/I term "file". Note that the conventions governing implicit references to MAD/I files dictate that "F" and "F " name the same MAD/I file while "F(1,10)" names a different MAD/I file although all three forms use the same MTS file. A FDUB is acquired from MTS each time a MAD/I file is opened with a data set name specification.

Examples: (using MTS conventions)

```
'OPEN'('DATA SET' "*SYSMAC",'END OF FILE'
       MACEND),MACLIB
'OPEN'(,MACEND,'DATA SET' "*SYSMAC"),MACLIB
```

are equivalent and open the file MACLIB which uses the data set consisting of the MTS file "*SYSMAC".

```
'READ DATA'('DATA SET' "*SOURCE*")
```

performs a data-directed input transmission using the data set consisting of the MTS FDname *SOURCE*, which is usually the user's terminal (or batch stream.) The file associated with the data set name specification "*SOURCE*" will be implicitly opened the first time it is referenced through the execution of an input/output statement which specifies it.

Part I -- Description of the MAD/I Language

6.3.3   Character-string Specification

      The character-string specification allows a character-
string expression to be used as if it were a data set containing
one record. A character-string specification is specified
through the 'STRING DATA SET' specification in the input/output
statement. This specification is a character-string expression.
For output transmission, it is restricted to a designator which
references a character-string.

      All implicit references to files by means of a character-
string specification will be considered as references to
different files.

Examples:

        'OPEN'('STRING DATA SET' "data string"),DATASTRING

opens the file DATASTRING which uses the data set consisting of
one record, the contents of the constant "data string".
DATASTRING can only be used for an input activity, since a
constant cannot be used as a designator.

        'DECLARE' S 'VARYING CHARACTER'(256)
        'OPEN'('STRING DATA SET' S),DATASTRING

opens the file DATASTRING which can be used for either an input
activity or an output activity. In either case, the data set is
considered to have a capacity of only one record.

        'DECLARE' S 'VARYING CHARACTER'(256)
        'WRITE DATA'('STRING DATA SET' S),data-list

performs a data-directed output transmission using the variable
S as the data set. The file associated with the character-
string specification S will be implicitly opened each time it is
referenced through the execution of an input/output statement
which specifies it. Thus the character-string specification S
can be used repeatedly, but only one record can be read or
written with it during each execution of an input/output
statement.

6.3.4 Entry-name Specification

The entry-name specification allows a data set to be defined in terms of two procedures, one which is called for every input record, the other which is called for every output record.   An entry-name specification is specified through the 'ENTRIES' specification in the input/output statement.   This specification consists of a variable of 'ENTRY NAME' mode or a parenthesized list of two variables of 'ENTRY NAME' mode.   The first (or only) variable is called once for every input record required. The input record must be returned as an expression of 'VARYING CHARACTER' mode.   The second variable is called once for every output record. The call includes one parameter, the contents of the output record in a variable of 'VARYING CHARACTER' mode. An end-of-file or end-of-volume condition can be returned through a return index of 1.

All implicit references to files which satisfy the following three rules will be considered as references to the same file:

1. All references are by means of an entry-name specification.
2. All the variables of 'ENTRY NAME' mode for reading compare as equal or all are missing.
3. All the variables of 'ENTRY NAME' mode for writing compare as equal or all are missing.

Examples:

```
'OPEN'('ENTRIES' IN),PROCFILE
'OPEN'('ENTRIES'(IN,OUT)),PROCFILE
'OPEN'('ENTRIES'(,OUT)),PROCFILE
```

all open the file PROCFILE which calls the procedures IN and OUT for input activity and output activity respectively. Omitting the input procedure (example 3) causes an end-of-file condition on input transmission requests; omitting the output procedure (example 1) causes an end-of-volume condition on output transmission requests.

```
'READ DATA'('ENTRIES'(IN,OUT))
```

performs a data-directed input transmission using the data set associated with the entry-name specification (IN,OUT). The file associated with this specification will be implicitly opened the first time it is referenced through the execution of an input/output statement which specifies it.

6.3.5 Default Specification

A data set is associated with a file by default if none of the previous four ways of specifying the data set has been used in the input/output statement. The default data set for input is that associated with the system standard input unit. The default data set for output is that associated with the system standard output unit.

All implicit references to the default _input_ are considered to be references to the _same_ file. All implicit references to the default _output_ are considered to be references to the _same_ file, but _different_ from the file assumed for default input.

In MTS, the default data set for input is that associated with the logical I/O unit SCARDS; the default data set for output is that associated with SPRINT.

In OS, the default data set for input is that associated with the ddname SYSIN; the default data set for output is that associated with SYSPRINT.

Examples: (using MTS conventions)

        'OPEN'('END OF FILE' A,'END OF VOLUME' B),FNAME
        'OPEN' (,A,B),FNAME

are equivalent and open the file FNAME which uses the system standard input unit (SCARDS) for input and the system standard output unit (SPRINT) for output.

        'READ DATA'

performs a data-directed input transmission using the default input file which is associated with the logical I/O unit SCARDS. The default input file will be implicitly opened the first time it is referenced through the execution of an input/output statement which specifies it.

6.4 File Attributes

Each file has a collection of attributes associated with it. For explicitly opened files, the attributes and their values can be specified in the 'OPEN' statement. Attributes which are omitted are given default values. For implicitly opened files, all attributes are given default values. The value of most file attributes can be overridden in any input/output statement for the duration of the execution of that statement.

6.4.1 Data set Associated with the File

The most important attribute of a file is the data set name used by the file and the manner in which this name was specified. This attribute has been described in Section 6.3. This attribute cannot be overridden in an input/output statement.

6.4.2 End-of-file File Attribute

The end-of-file file attribute is specified through the 'END OF FILE' specification of an input/output statement and has as its value an entry-name variable. This entry-name is called whenever an end-of-file condition is sensed from the data set associated with the file in response to an input request. The default end-of-file attribute value is the system subroutine which terminates execution. The end-of-file file attribute can be overridden in an input/output statement. In this case its value can be either an entry-name or a transfer-point.

In MTS, the default end-of-file attribute value is the system subroutine SYSTEM.

Examples:

        'OPEN' ("SPRINT",MACEND),FNAME
        'OPEN' ('END OF FILE' MACEND,'UNIT' "SPRINT"),FNAME
        'READ DATA' (FNAME,NEWEND)
        'READ DATA' ('END OF FILE' NEWEND,'UNIT' FNAME)
        'READ DATA' ('END OF FILE' NEWEND)
        'READ DATA' (,NEWEND)

6.4.3 End-of-volume File Attribute

The end-of-volume file attribute is specified through the 'END OF VOLUME' specification of an input/output statement and has as its value an entry-name variable. This entry-name variable is called whenever an end-of-volume condition is sensed

from the data set associated with the file in response to an
output request. The default end-of-volume attribute value is
the system subroutine which terminates execution. The end-of-
volume attribute can be overridden in an input/output statement.
In this case its value can be either an entry-name or a
transfer-point.

     In MTS, the default end-of-volume attribute value is the
system subroutine SYSTEM.

Examples:

          'OPEN'("SPRINT",,MACEND),FNAME
          'OPEN'('END OF VOLUME' MACEND,'UNIT' "SPRINT"),FNAME
          'WRITE DATA'(FNAME,NEWEND),data-list
          'WRITE DATA'('END OF VOLUME' NEWEND,'UNIT' FNAME),data-
               list
          'WRITE DATA'('END OF VOLUME' NEWEND),data-list
          'WRITE DATA'(,NEWEND),data-list

6.4.4 Error File Attribute

     The error file attribute is specified through the 'ERROR'
specification of an input/output statement and has as its value
an entry-name variable. This entry-name variable is called
whenever an error condition is sensed from the data set
associated with the file in response to an input or output
request. The default error attribute value is the system
subroutine which terminates execution abnormally. The error
attribute can be overridden in an input/output statement. In
this case its value can be either an entry-name or a transfer-
point.

     In MTS, the default error attribute value is the system
subroutine ERROR.

Examples:

          'OPEN'("SPRINT",,,MACERR),FNAME
          'OPEN'('ERROR' MACERR,'UNIT' "SPRINT"),FNAME
          'WRITE DATA'(FNAME,,NEWERR),data-list
          'READ DATA'(FNAME,,NEWERR)
          'WRITE DATA'('ERROR' NEWERR,'UNIT' FNAME),data-list
          'READ DATA'('ERROR' NEWERR,'UNIT' FNAME)
          'WRITE DATA'('ERROR' NEWERR),data-list
          'READ DATA'('ERROR' NEWERR)
          'WRITE DATA'(,,NEWERR),data-list
          'READ DATA'(,,NEWERR)

6.4.5 Maximum-length File Attribute

The maximum-length file attribute is specified through the
'MAX LENGTH' specification of an input/output statement and has
as its value an arithmetic expression or a parenthesized list of
two arithmetic expressions. If only one expression is given,
its value, truncated to the next lower integer value, is taken
as the maximum input and output record length in bytes. If two
expressions are given, the value of the first, truncated to the
next lower integer value, is taken as the maximum input record
length in bytes; the value of the second, similarly truncated,
is taken as the maximum output record length in bytes. The
default maximum-length file attribute values are the maximum
input and output record lengths allowed for the data set
associated with the file. The maximum-length file attribute can
be overridden in an input/output statement.

In MTS, the default maximum-length attribute values are the
maximum input and output record lengths as returned by the
subroutine GDINFO.

Examples:

```
        'OPEN'("SPRINT",'MAX LENGTH' 133),FNAME
        'OPEN'("SPRINT",'MAX LENGTH'(255,71)),FNAME
        'WRITE DATA'(FNAME,'MAX LENGTH' NEWLN),data-list
        'READ DATA'('MAX LENGTH' NEWLN)
```

6.4.6 Echo File Attribute

The echo file attribute is specified through the 'ECHO'
specification of an input/output statement and has as its value
any operand acceptable as a unit specification. Every
input/output transmission using the file is echoed on the unit
specified by the echo file attribute. The default echo file
attribute value is no echoing. The echo attribute can be
overridden in an input/output statement.

Examples: (using MTS conventions)

```
        'OPEN'("SPRINT",'ECHO' "SERCOM"),FNAME
        'READ DATA'('ECHO' "SPRINT")
        'READ DATA'('ECHO' FNAME)
```

Part I -- Description of the MAD/I Language

6.5 Miscellaneous Input/Output Specifications

The miscellaneous input/output specifications are used to specify both required and optional information within an input/output statement.

6.5.1 Format Specification

A format can be specified through a 'FORMAT' specification in an input/output statement. The format is used in format-directed transmission to control the form and conversion of data. This specification is done through a character-string expression, whose value is the format. The value of this expression is interpreted in a system-dependent fashion; there is no specification of a format language as a part of MAD/I.

In MTS, IOH360 is used as the format interpreter. The character-string expression must be a valid format as described in the IOH360 description in the MTS manual.

Examples: (using MTS conventions)

```
'WRITE'("' X=',F10.0,' X*X=',F10.0*"),X,X*X
'WRITE'('UNIT' "SERCOM",'FORMAT' "' FILE ',C,' HAS BEEN
     CREATED.'*"),FNAME
```

6.5.2 Line Specification

A line specification is used to perform random accesses to a data set. The line is specified via the 'LINE' specification, which specifies an arithmetic expression. The value of this expression is interpreted in a system-dependent fashion to determine the position in the data set at which the input/output activity of the current statement is to begin. Further input/output activity will be conducted in a sequential fashion until the next occurrence of a line specification.

In MTS, the line specification can be used for line files or sequential files. For line files, the value of the arithmetic expression is interpreted as the line number, multiplied by 1000, of the line to be next read or written. That is, the expression must have a value of 1500 to read or write beginning at line 1.5 of the file. This is the same value as used by the MTS input/output subroutines, such as SCARDS. For sequential files, the value of the arithmetic expression must be a value returned by a 'LAST LINE' specification in a previous input/output statement. This value is used internally to retrieve the corresponding note-point information. Both the read and write pointers are updated with the appropriate values.

Examples: (using MTS conventions)

          'READ'("I5*",0,'LINE' 1000),NUMB
          'WRITE'("I5*",0,'LINE' A+B),NUMB

6.5.3 Last-line Specification

A last-line specification is used to record the current
position in a data set so that a file can later be re-positioned
to that position in the data set. This is specified via the
'LAST LINE' specification, which consists of a designator for an
arithmetic value. The input/output system returns a value which
can be used in the 'LINE' specification to position the data
set. This returned value is treated in a system-dependent
fashion.

In MTS, the last-line specification can be used for any
data set. For line files, the value returned is the line number
of the last record read or written by this statement, multiplied
by 1000. That is, the value 1500 is returned if 1.5 was the
line number of the last record read or written by the statement.
This is the same value as used by the MTS input/output
subroutines, such as SCARDS. For sequential files, the value
returned is a code used internally to retrieve the note-point
information corresponding to the last record read or written.
For all other types of data set organization, a pseudo line
number is returned as computed by MTS.

Examples: (using MTS conventions)

          'READ'("I5*",0,'LAST LINE' LLINE),NUMB
          'WRITE'("I5*",'LAST LINE' LNUM),NUMB

6.5.4 Last-length Specification

A last-length specification is used to obtain the length,
in bytes, of the last record read or written by the input/output
statement. This is specified via the 'LAST LENGTH'
specification, which consists of a designator for an arithmetic
value.

Examples:

          'READ UNCONVERTED'(0,'LAST LENGTH' N),ARRAY
          'WRITE DATA'('LAST LENGTH' LEN),data-list

          Part I -- Description of the MAD/I Language

6.6 Input/output Specification Summary

The following table summarizes all the possible input/output specifications and the possible modes of their value expressions.

Keyword	File Atr?	Designator?	Permissible Modes
'DATA SET'	Yes[4]	No	Character-string
'ECHO'	Yes	No	Arithmetic Character-string Filename
'END OF FILE'	Yes	No	Entry-name Transfer-point [1]
'END OF VOLUME'	Yes	No	Entry-name Transfer-point [1]
'ENTRIES'	Yes[4]	No	Entry-name Two entry-name [2]
'ERROR'	Yes	No	Entry-name Transfer-point [1]
'FORMAT'	No	No	Character-string
'LAST LENGTH'	No	Yes	Arithmetic
'LAST LINE'	No	Yes	Arithmetic
'LINE'	No	No	Arithmetic
'MAX LENGTH'	Yes	No	Arithmetic Two arithmetic [2]
'STRING DATA SET'	Yes[4]	Yes[3]	Varying-character
'UNIT'	Yes[4]	No	Arithmetic Character-string Filename

(1) Transfer-point expressions cannot be used in 'OPEN' statements.
(2) Two expressions are represented as a parenthesized list of two expressions.
(3) Need not be a designator for input activity.
(4) These specifications are used to denote the file to be used; hence, at most one of these can be given per input/output statement.

6.7   Data-lists

A data-list is used to specify the designators to which
data are to be assigned (for input activity) and the data values
to be transmitted (for output activity.)   The elements of a
data-list may be either block-elements or expressions. For
input activity, the expressions are restricted to designators.
For example, the data-list

A, X+3, C

is valid for output activity but not for input activity, because
X+3 is not a designator.   In either case, it should be
understood that expressions include embedded statements.   For
input activity, further references involving designators earlier
used as data-list expressions refer to the new value just read.
For example, N,A(N) uses the new value of N in forming the
reference to A(N).

6.7.1   Block Elements

A block-element is a pair of subscripted elements from the
same array separated by an ellipsis (without commas).   The
subscripts may be arbitrarily complex. An example of a block-
element is:

A(I,J)...A(I+3,K)

The block-element represents all the elements of the array, from
the   first-named   element   through   the   second-named   element,
sequencing through the elements in the order determined by the
array sequencing rule (Sec. 3.1.2.1).   For example, if we have
declared

A 'FIXED ARRAY' (-1...1,2,0...2)

then

A(0,1,1)...A(0,2,2)

represents the five elements

A(0,1,1), A(0,1,2), A(0,2,0), A(0,2,1), A(0,2,2) .

The number of array elements represented by a block-element   can
vary   during   execution   as   the   subscript   values   vary.   For
example, B(1)...B(N), where B has been declared   an   array   with
one dimension, represents N array elements.

Part I -- Description of the MAD/I Language

6.7.2 Array Expressions

An expression whose result is of an array mode represents
all the elements of the array, sequencing through the elements
in order. For example, if we have declared

C 'FIXED ARRAY'(-1...1,2)

then the use of C as an expression in a data-list represents the
six elements

C(-1,1), C(-1,2), C(0,1), C(0,2), C(1,1), C(1,2) .

6.7.3 Component-structure Expressions

An expression whose result is a component structure
represents all the elements of the component structure from
left-to-right in the same order as declared. For example, if we
have declared

D 'COMPONENT STRUCTURE'(∂A 'INTEGER',∂B 'FIXED ARRAY'(2))

then the use of D as a data-list expression represents the three
items

D$∂A, D$∂B(1), D$∂B(2) .

6.7.4 Unsupported Modes

Expressions whose result is one of the following modes
cannot be used as data-list expressions: 'ALTERNATE', 'BIT',
'ENTRY POINT', 'FILE NAME', and 'TRANSFER POINT'.

6.7.5 Embedded Statements

An embedded statement can be used as an expression in a data-list. For prefix statements, the expressions in their scope (i.e., the expressions following the comma in the short form or the expressions delimited by semicolons in the long form), will be called scope expressions. In the execution of the embedded statement in the data-list, any scope expressions which appear in a 'LIST' statement are treated as a part of the data-list. For an input activity, the scope expressions of a 'LIST' statement are restricted to designators.

Examples:

The data-list

        N, ('FOR' I := 1,1,I>N, 'LIST' X(I))

is equivalent to the data-list

        N, X(1)...X(N) .

The data-list

        N,('FOR' I := 1,1,I>N; 'LIST' X(I),Y(I); 'IF' I>1,
            'LIST' Z(I) 'END')

is equivalent to

        N,X(1),Y(1),X(2),Y(2),Z(2),X(3),Y(3),Z(3), --
            ,X(N),Y(N),Z(N) .

6.8 Syntax of the Input/output Statements

An input/output statement (other than 'CLOSE') consists of a keyword, followed by an optional parenthesized specification list, optionally followed by a comma and a data-list. A close statement consists of the keyword 'CLOSE' followed by a file-name expression.

   I/O-statement = I/O-keyword [I/O-spec-list] [, data-list]

   close-statement = 'CLOSE' filename-expression

Examples:

         'OPEN'("SPRINT",MACEND),FNAME
         'READ DATA'
         'READ',A,B,C
         'CLOSE' FNAME

The permissible input/output statement keywords are: 'OPEN', 'READ', 'READ DATA', 'READ UNCONVERTED', 'WRITE', 'WRITE DATA', and 'WRITE UNCONVERTED'.

   I/O-keyword = 'OPEN' | 'READ' | 'READ DATA' | 'READ
                  UNCONVERTED' | 'WRITE' | 'WRITE DATA' |
                  'WRITE UNCONVERTED'

An input/output specification list consists of a parenthesized list of one or more specifications which can be given in a positional or a keyword form, or a mixture of both. For each input/output statement, the input/output specifications are each assigned a position in the list, from most-commonly-used specification (on the left) to least-commonly-used specification (on the right). A specification can be given in positional form by putting its expression in the appropriate position in the input/output specification list. Specifications can be skipped over in the positional form by using successive commas. Positional specifications cannot be used to the right of the first keyword specification in the list. A specification can be given in keyword form by preceding its expression by the appropriate keyword. A keyword specification can be given in any position in the list. The syntax is as follows:

```
I/O-spec-list = ( I/O-keyword-spec-list )
              | ( I/O-positional-spec-list )
              | ( I/O-positional-spec-list ,
              I/O-keyword-spec-list )

I/O-positional-spec-list = list , I/O-spec-expr

I/O-keyword-spec-list = list , ( I/O-spec-keyword
                            I/O-spec-expr }

I/O-spec-keyword = 'DATA SET' | 'ECHO' | 'END OF FILE' |
                   'END OF VOLUME' | 'ENTRIES' | 'ERRCR' |
                   'FORMAT' | 'LAST LENGTH' | 'LAST LINE'
                   | 'LINE' | MAX LENGTH' | 'STRING DATA
                   SET' | 'UNIT'

I/O-spec-expr = expression
              | ( expression , expression )
              | ( , expression )
```

Examples:

```
("SCARDS",ENDFILE)
("SCARDS",ENDFILE,'MAX LENGTH' 72)
('MAX LENGTH' (72,132))
("SCARDS",,GOERR)
```

A data-list consists of a list of expressions and block-elements, separated by commas. Data-lists have been discussed in Section 6.7.

```
data-list = list', { expression | block-element }

block-element = array-element-desig ... array-element-desig
```

6.9 Input/Output Statements

A brief description of the input/output statements is given
below. Each descriptive section begins with the statement
prototype followed by a list, in positional order, of the
acceptable specification elements (which may, of course, take
default values).

6.9.1 File Specification

Statement Prototype:
        'OPEN' [I/O-spec-list] , filename-designator

Allowable Specification Keywords: 'UNIT', 'END OF FILE', 'END OF
    VOLUME', 'ERROR', 'ECHO', 'MAXLENGTH', 'DATA SET', 'STRING
    DATA SET', 'ENTRIES'

The file referenced by the filename-designator is
explicitly opened. The values of its file attributes are
determined by the I/O-spec-list. Those attributes which are not
given values take on default values. All file attributes (other
than the data set associated with the file) can be overridden in
input/output statements which reference the file. A file which
has been explicitly opened can be used in the unit specification
of all input/output statements other than 'OPEN' until the file
is closed through a 'CLOSE' statement.

Examples:

        'OPEN'(0,MACEND),MACLIB
        'OPEN'('END OF FILE' MACEND, 'UNIT' 0),MACLIB
        'OPEN'(,MACEND,'DATA SET' "*SYSMAC"),MACLIB
        'OPEN'('ENTRIES'(IN,OUT)),PROCFILE
        'OPEN',DEFAULTFILE

Statement Prototype: 'CLOSE' filename-expression

The explicitly opened file specified by the filename-
expression is closed. The filename-expression cannot be used in
any further input/output statements without being opened again.
No other copies of the value of the filename-expression can be
used in further input/output statements, even if the file is
opened again. If a data set name specification ('DATA SET') was
used when the file was opened, the system in which the MAD/I
program is being run is notified that this usage of the data set
has ceased. The system is then free to close the data set when
it feels that is appropriate.

Examples:

```
'CLOSE' MACLIB
'CLOSE' FILEARRAY(I+3)
```

6.9.2 Data-Directed I/O

Data-Directed Input

Statement Prototype: 'READ DATA' [I/O-spec-list] [, data-list]

Allowable Specification Keywords: 'UNIT', 'END OF FILE', 'ERROR', 'LINE', 'LAST LINE', 'LAST LENGTH', 'ECHO', 'MAX LENGTH', 'DATA SET', 'STRING DATA SET', 'ENTRIES'

    This statement causes a data-directed input transmission. The format of acceptable input records is discussed in Section 6.2.1. If the data-list is given, the designators allowed on the input records are restricted to those which reference the variables specified in the data-list. If no data-list is given, any designator which is valid within the block containing the 'READ DATA' statement can be given in the input records. All variables known within the block which can be specified in the input records as described above will automatically be entered (at compile time) in the run-time symbol table.

Examples:

```
'READ DATA'
'READ DATA'('ECHO' "SPRINT")
'READ DATA'('DATA SET' "INITVALUES")
'READ DATA', A,B,COMPLXN,Z
```

each could be used to read the record:

$$A:=-3.2, \ B:="S", \ COMPLXN\$ \partial R:=1.5, \ Z(2):=1,,5(2) ;$$

The first three examples would force all the variables known within the block containing the 'READ DATA' statement into the run-time symbol table, while the last example would force only A, B, COMPLXN, and Z into the run-time symbol table.

Data-Directed Output

Statement Prototype: 'WRITE DATA' [I/O-spec-list] , data-list

Allowable Specification Keywords: 'UNIT', 'END OF VOLUME', 'ERROR', 'LINE', 'LAST LINE', 'LAST LENGTH', 'ECHO', 'MAX LENGTH', 'DATA SET', 'STRING DATA SET', 'ENTRIES'

Part I -- Description of the MAD/I Language

This statement causes a data-directed output transmission. The format of the output records produced is discussed in Section 6.2.1. Symbol table entries for each element in the data-list will automatically be entered in the run-time symbol table.

Examples:

        'WRITE DATA',X+3,A

would produce an output record like:

        *** = 10, A = -3.2;

Also,

        'WRITE DATA'('DATA SET' "NEWVALUES"),Z(2)...Z(5)

would produce an output record like:

        Z(2) = 1.5, 3.6, -10.2, 8.63;

6.9.3 List-Directed I/O

List-Directed Input

Statement Prototype: 'READ' [I/O-spec-list] , data-list

Allowable Specification Keywords: □, 'UNIT', 'END OF FILE', 'ERROR', 'LINE', 'LAST LINE', 'LAST LENGTH', 'ECHO', 'MAX LENGTH', 'DATA SET', 'STRING DATA SET', 'ENTRIES'

This describes a list-directed input transmission. The □ represents the 'FORMAT' I/O specification. List-directed input/output is distinguished from format-directed input/output by the absence of the 'FORMAT' specification. The format of acceptable input records is discussed in Section 6.2.2. Symbol table entries for each data-list element will automatically be entered in the run-time symbol table.

Examples:

        'READ',N,M,X(1)...X(N)
        'READ'(,0),N,M,X(1)...X(N)
        'READ'('UNIT' 0),N,M,X(1)...X(N)

each could be used to read the record:

        4 , "CASE 1", 1.5, 3.2 , -.3, 16

List-Directed Output

Statement Prototype: 'WRITE' [I/O-spec-list] , data-list

Allowable Specification Keywords: n, 'UNIT', 'END OF VOLUME', 'ERROR', 'LINE', 'LAST LINE', 'LAST LENGTH', 'ECHO', 'MAX LENGTH', 'DATA SET', 'STRING DATA SET', 'ENTRIES'

    This describes a list-directed output transmission. The n represents the 'FORMAT' I/O specification. List-directed input/output is distinguished from format-directed input/output by the absence of the 'FORMAT' specification. The format of output records produced is discussed in Section 6.2.2. Symbol table entries for each data-list element will automatically be entered in the run-time symbol table.

Examples:

```
'WRITE', N,"VALUES ARE:",X(1)...X(N)
'WRITE'(,0), N,"VALUES ARE:",X(1)...X(N)
'WRITE'('UNIT' 0), N,"VALUES ARE:",X(1)...X(N)
```

each would produce a record like:

```
4 VALUES ARE: 1.5 3.2 -0.7 16.0
```

6.9.4 Format-Directed I/O

Format-Directed Input

Statement Prototype: 'READ' I/O-spec-list , data-list

Allowable Specification Keywords: 'FORMAT', 'UNIT', 'END OF FILE', 'ERROR', 'LINE', 'LAST LINE', 'LAST LENGTH', 'ECHO', 'MAX LENGTH', 'DATA SET', 'STRING DATA SET', 'ENTRIES'

    This describes a format-directed input transmission, as described in Section 6.2.3. Format specifications themselves are discussed in Section 6.5.1.

Examples:

```
'READ' ("I5*"),N
'READ' ("I5*",0),N
```

each could be used to read the record:

```
ooo3o
```

Part I -- Description of the MAD/I Language

where each □ represents a blank.

Format-Directed Output

Statement Prototype: 'WRITE' I/O-spec-list [, data-list]

Allowable Specification Keywords: 'FORMAT', 'UNIT', 'END OF VOLUME', 'ERROR', 'LINE', 'LAST LINE', 'LAST LENGTH', 'ECHO', 'MAX LENGTH', 'DATA SET', 'STRING DATA SET', 'ENTRIES'

This describes a format-directed output transmission, as described in Section 6.2.3. Format specifications themselves are discussed in Section 6.5.1. The first character of the output line may be treated as a logical carriage control, depending upon the system in which the MAD/I program is being run and the type of the data set organization.

Examples:

       'WRITE'("'&ENTER THE FILE NAME:'*")

will produce the output record:

       &ENTER THE FILE NAME:

In MTS, the "&" will be treated as a logical carriage control which suppresses a line-feed at the end of the line if the data set is a terminal.

       'WRITE'("'X=',F3.2*"),X
       'WRITE'("'X=',F3.2*",0),X

each would produce an output record like:

       X=315.52

6.9.5 Unconverted I/O

Unconverted Input:

Statement Prototype:
       'READ UNCONVERTED' [I/O-spec-list] , data-list

Allowable Specification Keywords: 'UNIT', 'END OF FILE', 'ERROR', 'LINE', 'LAST LINE', 'LAST LENGTH', 'ECHO', 'MAX LENGTH', 'DATA SET', 'STRING DATA SET', 'ENTRIES'

This statement causes an unconverted input transmission as described in Section 6.2.4. Exactly one record will be read. The record must have been written with a 'WRITE UNCONVERTED' statement. The data-list items must agree in mode with those specified in the 'WRITE UNCONVERTED' statement which produced the record. Symbol table entries for all the variables which are referenced in the data-list will automatically be entered in the run-time symbol table. Unconverted input/output is the most efficient type of transmission because conversion is not needed.

Examples:

           'READ UNCONVERTED', N,X(1)...X(N)
           'READ UNCONVERTED'(0), N,X(1)...X(N)

Unconverted Output:

Statement Prototype:
           'WRITE UNCONVERTED' [I/O-spec-list] , data-list

Allowable Specification Keywords: 'UNIT', 'END OF VOLUME', 'ERROR', 'LINE', 'LAST LINE', 'LAST LENGTH', 'ECHO', 'MAX LENGTH', 'DATA SET', 'STRING DATA SET', 'ENTRIES'

This statement causes an unconverted output transmission as described in Section 6.2.4. Exactly one record will be written. The record can be read only with a 'READ UNCONVERTED' statement whose data-list items agree in mode to those specified in the 'WRITE UNCONVERTED' statement which produced the record. Symbol table entries for all the data-list elements will automatically be entered in the run-time symbol table. Unconverted input/output is the most efficient type of transmission because conversion is not needed.

Examples:

           'WRITE UNCONVERTED', N,X(1)...X(N)
           'WRITE UNCONVERTED'(0), N,X(1)...X(N)
           'WRITE UNCONVERTED',N+3,M,X(1,1)...X(N+3,M)

Section 7: Program Structure

7.1 Block Structure

Like other languages such as PL/I and ALGOL 60, MAD/I
includes the concept of block structure.

There are two kinds of blocks: compound-statement blocks
and procedure blocks.

A compound-statement block is a 'BLOCK' statement
(Sec. 5.10). The block begins with the statement keyword
'BLOCK', ends with the corresponding end keyword 'END', and
contains all the intervening text. If the statement is labeled,
the label is not contained in the block.

A procedure block is a 'PROCEDURE' statement (Sec. 5.7.1).
The block begins with the statement keyword 'PROCEDURE', ends
with the corresponding end keyword ('END PROCEDURE' or 'END'),
and contains all the intervening text. If the statement is
labeled, the label is not contained in the block. Both the
short-form and the long-form 'PROCEDURE' statements are blocks.

Blocks may be properly nested; a block may contain other
blocks, which may in turn contain other blocks. We will say
that a portion of text T (a symbol, expression, or statement) is
internal to a block B, and that B properly contains T, if and
only if:

(1)   B contains T, and

(2)   there is no block C such that B contains C and C
      contains T.

Every MAD/I program is a block -- either a compound-
statement block or a procedure block. It is called the
outermost block, and is not internal to any block. Every other
block in the program is internal to exactly one block.

7.2 Scope of names

The block structure of a program provides a convenient framework in which to define the "scope of names" and a "re-naming convention". We shall take "names" to mean identifiers only, although these concepts could potentially be extended to operators and keywords as well.

The re-naming convention allows the same sequence of source-program characters (i.e., the same symbol) to be used to represent more than one name in the program, provided that the different usages are disjoint, so that each instance of the symbol represents a well-defined name.

Example:

```
'PROCEDURE' AA;
'INTEGER' I, J, K;
stmt-seq-A1;

    'PROCEDURE' BB;
    'INTEGER' I, J, L;
    stmt-seq-B;
    'END PROCEDURE';

stmt-seq-A2;
'END PROCEDURE'
```

In the above example, the four symbols I, J, K, and L are used to represent six different identifiers (names):

```
I  in block AA, but not block BB;
J  in block AA, but not block BB;
K  in block AA;
I  in block BB;
J  in block BB;
L  in block BB.
```

The scope of a name is the union of all portions of a program (or a linked set of programs) in which the name is "known"; i.e., all places where it may be used. (See also Section 3.3.) A name is "known" in a portion of text T if an instance in T of the symbol representing the name is recognized as an occurrence of that name.

Every name must be declared (explicitly or implicitly) in some block (see Sections 3.5-3.7). The scope of a given name N can be determined as follows (let S be the symbol representing N):

(1) The scope of N includes all text internal to the block B in which N is declared (B is the block to which the declaration of N is internal).

Part I -- Description of the MAD/I Language

(2)   If the scope of N includes declarations that N is 'NOTNEW'
      or 'GLOBAL', the scope of N is extended "outward"
      accordingly (see Sec. 3.3). Multiple declarations of N are
      permitted so long as they do not conflict, but only one
      mode declaration is allowed.

(3)   Let B1 be the smallest block containing all the scope of N
      (B1 either is B or contains B). The scope of N is now
      extended "inward" into all blocks internal to B1, and all
      blocks internal to those, etc., except that the scope of N
      is not extended into any block which properly contains the
      scope (or part of the scope) of any other name N'
      represented by the same symbol S.

(4)   Names declared 'EXTERNAL' or 'ACCESSIBLE' are called
      "external" names.   If two or more external names are
      represented by the same symbol, they are merged into a
      single name whose scope is the union of the individual
      scopes. The attributes of the names must not conflict.

7.3 Block structure at Run time

At run time, blocks are _activated_ (entered) and _terminated_ (exited) in a dynamic sequence determined by the order of execution of the program.

A compound-statement block is activated when control passes through the statement keyword ('BLOCK') for the block. It is terminated when control passes through the end keyword ('END') for the block, or when execution of a 'GO TO' statement transfers control to a point not in the block.

A procedure block is activated when any one of its entry points is called. It is terminated in any of the ways described in Section 5.7.3.

Recursive procedures have not yet been defined in the MAD/I language.

Section 8: Compile-Time Facilities

Sometimes it is useful to be able to perform operations
that result in some change in the source text. These operations
or computations are performed at compilation time and not at run
time. MAD/I has some facilities for performing compile-time
operations.

8.1 The 'SUBSTITUTE' Statement

The 'SUBSTITUTE' statement may be used to associate a given
symbol (other than a constant) with an arbitrary sequence of
symbols at translation time. The form of this statement is

'SUBSTITUTE' X S1 S2 ... Sn 'END SUBSTITUTE'

where X and S1 through Sn are legal MAD/I symbols and X is not a
constant. After the occurrence of this statement, each
occurrence of the symbol X will be replaced by the sequence of
symbols S1 through Sn.

Substitution would normally be used for representing either
repetitious portions of a program or some sequence occurring in
many parts of a program and changing from translation to
translation.

Note that S1 through Sn must be complete symbols. Also,
the context in which X occurs will in no way affect the
recognition of the symbols S1 through Sn.

Since the substitution of a symbol is effective only after
it has been defined by a 'SUBSTITUTE' statement, that symbol may
have had a different meaning (i.e., may have been a variable,
operator, constant, etc.) previously. Whenever a substitution
definition is assigned to a symbol, the previous meaning is
pushed down. Previous definitions of a symbol may be restored
by means of the 'POP SUBSTITUTE' statement, which has the form:

'POP SUBSTITUTE' X

This will cause the last previous meaning of X to be
restored. There is no limit on the number of redefinitions of a
symbol.

For example, the following program section:

        'SUBSTITUTE' SIZE 16 'END SUBSTITUTE';

        'SUBSTITUTE' PI 3.1415927 'END SUBSTITUTE';

        'SUBSTITUTE' + - 'END SUBSTITUTE';

        'SUBSTITUTE' IF 'IF' 'END SUBSTITUTE';

        'SUBSTITUTE' Q 3 * Q 'END SUBSTITUTE';

        'SUBSTITUTE' Q 2 * Q 'END SUBSTITUTE';

        'SUBSTITUTE' A 'FIXED ARRAY' (15,SIZE) 'END SUBSTITUTE';

        'SUBSTITUTE' TEMP := 'END SUBSTITUTE';

        'SUBSTITUTE' := = 'END SUBSTITUTE';

        'SUBSTITUTE' = TEMP 'END SUBSTITUTE';

        'POP SUBSTITUTE' TEMP ;

        'DECLARE' XYZ A ;

        IF MMM & DOW, EST = EST + 1 ;

        BOR1 = Q / PI;

        'POP SUBSTITUTE' Q;

        BOR2 = Q / PI;

is equivalent to

        'DECLARE' XYZ 'FIXED ARRAY' (15,16);

        'IF' MMM & DOW, EST := EST - 1;

        BOR1 := 2 * 3 * Q / 3.1415927;

        BOR2 := 3 * Q / 3.1415927;

8.2 The 'INCLUDE' form

The 'INCLUDE' form allows the programmer to specify, as a
part of the text of his source program, a place where more
source text may be obtained. The text so obtained is inserted
in place of the 'INCLUDE' form at compile time.

Syntax: 'INCLUDE' character-symbol

The character string in the character-symbol (Sec. 2.1.4.1)
specifies the location of the text to be included. The
'INCLUDE' form itself may occur anywhere in the source program
(except within a symbol or comment) -- it is not considered a
statement. The included text is obtained as a sequence of
characters, and is scanned like any other portion of source
text; it replaces the 'INCLUDE' form which specified it, and
should therefore be syntactically valid in the context of the
'INCLUDE' form. Included text may contain further 'INCLUDE'
forms.

The character string in the character-symbol is taken as a
data set name, and is interpreted in a system-dependent fashion
(see Section 6.3.2).

Example: 'INCLUDE' "DEFPACKAGE"

Section 9:   Definitional Facility

This section has not yet been written. Facilities are planned which will allow the programmer to define new data types, new operations, new operators, and new statements. New constructs would be defined either in terms of existing constructs (pre-defined or user-defined) or in terms of an assembler-like language.

The feasibility of an effective definitional facility has already been established by actual experiments with MAD/I. (See the memorandum by Srodawa which is cited in the Preface.) It remains to design and implement a clean mechanism which allows the user to express his definitions in a reasonable way. This requires more research.

One of the authors (Springer) is now writing a doctoral dissertation which describes an experimental definitional facility based on MAD/I.

Section 10: Example MAD/I Programs

10.1 Procedures CALLSQRT and SQRT

This example shows two MAD/I procedures, CALLSQRT and SQRT.
CALLSQRT is the "main" program and calls upon the procedure
SQRT. There is no main program declaration; CALLSQRT becomes
the main program by being the first program executed by the
operating system. The default mode is 'FLOATING SHORT' since it
is not otherwise declared. The procedure CALLSQRT reads a
number, then prints the number entered followed by the value
returned by SQRT. The procedure SQRT computes the square root
of its argument using a Newton-Raphson approximation technique.

```
          'PROCEDURE' CALLSQRT.;
CALLSQRT: 'WRITE'("'&ENTER X:'*");
          'READ' ("WF*"), X;
          'WRITE' ("' X=',WF,' SQRT OF X=',WF*"),X,SQRT.(X);
          'GO TO' CALLSQRT
          'END'
```

```
          'PROCEDURE' SQRT.(X);
          'PRESET' EPS := .0001;
SQRT:     'IF' X=0. | X=1., 'RETURN' X;
          Y := X;
LOOP:     Z := (Y+X/Y)/2.;
          'IF' .ABS.(Y-Z) < EPS, 'RETURN' Z;
          Y := Z;
          'GO TO' LOOP
          'END'
```

The following is a sample run of the procedures CALLSQRT
and SQRT. The numbers following "ENTER X:" are input data typed
by the user.

```
ENTER X:  100.0
X=    100.0000 SQRT OF X=     10.0000
ENTER X:  1.0
X=      1.0000 SQRT OF X=      1.0000
ENTER X:  0.
X=       .0000 SQRT OF X=       .0000
ENTER X:  4.0
X=      4.0000 SQRT OF X=      2.0000
ENTER X:  ¢
```

**** ALL INPUT DATA HAS BEEN PROCESSED - AT LOCATION 500788

The two independent procedures CALLSQRT and SQRT can be combined into one program by making SQRT internal to CALLSQRT. SQRT must be declared 'ACCESSIBLE' if it is to be referenced in other programs. The sample run of this program is identical to the previous sample run.

```
          'PROCEDURE' CALLSQRT.;
CALLSQRT: 'WRITE'("'&ENTER X:'*");
          'READ' ("WF*"), X;
          'WRITE' ("' X=',WF,' SQRT OF X=',WF*"),X,SQRT.(X);
          'GO TO' CALLSQRT;

          'PROCEDURE' SQRT.(X);
          'PRESET' EPS := .0001;
SQRT:     'IF' X=0. | X=1., 'RETURN' X;
          Y := X;
LOOP:     Z := (Y+X/Y)/2.;
          'IF' .ABS.(Y-Z) < EPS, 'RETURN' Z;
          Y := Z;
          'GO TO' LOOP
          'END'
          'END'
```

10.2 Procedures HASHTEST and HASH

The procedure HASH maintains a hashed symbol table. It is
called with one argument, the 'CHARACTER'(8) symbol to be
hashed. HASH then computes a key with a value ranging from 0
through 7 which is the hash of the symbol name. The operator
.AS. is used to treat the symbol as two integers in the
computation of the key.   Finally, the appropriate thread is
searched for a symbol table entry having the argument as its
name.   If no such entry is found, a new symbol table entry is
allocated using the 'ALLOCATE' statement and inserted at the
head of the appropriate thread. HASH returns the pointer to the
requested symbol table entry.

The procedure HASHTEST requests a symbol as input, calls
HASH with the symbol as the argument, and prints the pointer
returned and the contents of the symbol table entry.   HASHTEST
is the main program.

```
          'PROCEDURE' HASHTEST.;
          'DECLARE' 'DEFAULT' 'INTEGER';
          'DECLARE' HASH 'ENTRY POINT' 'POINTER';
          'DECLARE' PTR 'POINTER';
          'DECLARE' SYMENT 'BASED' 'COMPONENT STRUCTURE'(
            'POINTER',
            'CHARACTER'(8),
            'INTEGER',
            'BIT'(8) );
          'DECLARE' SYMBOL 'CHARACTER'(8);
HASHTEST: 'WRITE'("'&ENTER NEXT SYMBOL:'*");
          'READ'("C8.8*"), SYMBOL;
          PTR := HASH.(SYMBOL);
          SYMENT .ALLOC. PTR;
          'WRITE'("'SYMBOL TABLE ENTRY AT: ',X8.4,' PTR=',
            X8.4,' NAME=',C8.8*"), PTR,
            SYMENT(1),SYMENT(2);
          'GO TO' HASHTEST;
          'END'
```

```
            'PROCEDURE' HASH. (SYMBOL);

            'DECLARE DEFAULT' 'INTEGER';
            'DECLARE' SYMBOL 'CHARACTER'(8);
            'DECLARE' HASH 'ENTRY POINT' 'POINTER';
            'DECLARE' SYMENT 'BASED' 'COMPONENT STRUCTURE'(
              'POINTER',           << NEXT SYMBOL >>
              'CHARACTER'(8),      << SYMBOL NAME >>
              'INTEGER',           << STORAGE ALLOC >>
              'BIT'(8) );          << CLASS MODE >>
            'DECLARE' THREADS 'FIXED ARRAY'(0...6) 'POINTER';
            'DECLARE' FINGER 'POINTER';
            'DECLARE' NAMES 'FIXED ARRAY'(2) 'INTEGER';
            'PRESET' THREADS := 7('NULL PT');

HASH:       (NAMES .AS.('CHARACTER'(8))) := SYMBOL;
            KEY := .ABS.((NAMES(1)+NAMES(2)).REM.7);
            'WRITE' ("'**** KEY=',I*"),KEY;
            FINGER := THREADS(KEY);
LOOP:       'IF' FINGER = 'NULL PT';
              'ALLOCATE' SYMENT;
              SYMENT(1) := THREADS(KEY);
              THREADS(KEY) := .PT. SYMENT;
              SYMENT(2) := SYMBOL;
              SYMENT(4) := SYMENT(3) := 0;
            'ELSE';
              SYMENT .ALLOC. FINGER;
              'IF' SYMBOL = SYMENT(2), 'GO TO' FOUND;
              FINGER := SYMENT(1);
              'GO TO' LOOP
            'END';
FOUND:      'RETURN' .PT. SYMENT
            'END'
```

The following is a sample run of the procedures HASHTEST and HASH.

```
    ENTER NEXT SYMBOL:  a
**** KEY=       1
SYMBOL TABLE ENTRY AT: 00500068 PTR=00000000 NAME=A
    ENTER NEXT SYMBOL:  b
**** KEY=       2
SYMBOL TABLE ENTRY AT: 00500080 PTR=00000000 NAME=B
    ENTER NEXT SYMBOL:  c
**** KEY=       3
SYMBOL TABLE ENTRY AT: 00500098 PTR=00000000 NAME=C
    ENTER NEXT SYMBOL:  d
**** KEY=       4
SYMBOL TABLE ENTRY AT: 005000B0 PTR=00000000 NAME=D
    ENTER NEXT SYMBOL:  e
```

Part I -- Description of the MAD/I Language

```
**** KEY=        5
SYMBOL TABLE ENTRY AT: 00500C68 PTR=00000000 NAME=E
 ENTER NEXT SYMBOL:  f
**** KEY=        6
SYMBOL TABLE ENTRY AT: 00500C80 PTR=00000000 NAME=F
 ENTER NEXT SYMBOL:  g
**** KEY=        0
SYMBOL TABLE ENTRY AT: 00500C98 PTR=00000000 NAME=G
 ENTER NEXT SYMBOL:  h
**** KEY=        1
SYMBOL TABLE ENTRY AT: 00500CB0 PTR=00500068 NAME=H
 ENTER NEXT SYMBOL:  i
**** KEY=        2
SYMBOL TABLE ENTRY AT: 00500CC8 PTR=00500080 NAME=I
 ENTER NEXT SYMBOL:  a
**** KEY=        1
SYMBOL TABLE ENTRY AT: 00500068 PTR=00000000 NAME=A
 ENTER NEXT SYMBOL:  h
**** KEY=        1
SYMBOL TABLE ENTRY AT: 00500CB0 PTR=00500068 NAME=H
 ENTER NEXT SYMBOL:  aardvark
**** KEY=        4
SYMBOL TABLE ENTRY AT: 00500CE0 PTR=005000B0 NAME=AARDVARK
 ENTER NEXT SYMBOL:  quail
**** KEY=        0
SYMBOL TABLE ENTRY AT: 00500CF8 PTR=00500C98 NAME=QUAIL
 ENTER NEXT SYMBOL:  wunerful
**** KEY=        2
SYMBOL TABLE ENTRY AT: 00500D10 PTR=00500CC8 NAME=WUNERFUL
 ENTER NEXT SYMBOL:  a
**** KEY=        1
SYMBOL TABLE ENTRY AT: 00500068 PTR=00000000 NAME=A
 ENTER NEXT SYMBOL:  ¢

 **** ALL INPUT DATA HAS BEEN PROCESSED - AT LOCATION 5009E0
```

PART II -- USER'S GUIDE FOR MAD/I IN MTS

Section 11: The Compiler in MTS Public File *MAD1

Contents: The object modules which make up the MAD/I
                 compiler.

Purpose: To compile MAD/I programs.

Usage: The compiler is invoked by a RUN command,
                 specifying *MAD1 as the object file.

Logical I/O units referenced:

            SCARDS -   The source program to be compiled.

            SPRINT -   The compiler listings and
                       diagnostics.

            SPUNCH -   The resulting object module. This
                       can be controlled by the DECK
                       option.

Examples: $RUN *MAD1
                 (SCARDS, SPRINT, SPUNCH default to *SOURCE*
                 SINK, *PUNCH*, respectively)

                 $RUN *MAD1 SPUNCH=-F1 PAR=NOSOURCE

Description: See Part I of this manual for a description of
                 the MAD/I Language.

     Compiler options can be passed by the optional PAR= field
on the RUN command. This field must be the last in the sequence
of specifications on the RUN command. The PAR= field consists
of a list of option specifications separated by blanks or
commas. Many of the option keywords have abbreviations. Some
options have pairs of alternative keywords of the forms
"option", "NOoption". In each case, the "option" keyword
requests a service, and the "NOoption" keyword rejects the
service. Each option has a default. The default value for some
options depends upon whether the compiler is being run in batch
or from a terminal. In case of conflicting options, the right-
most option specification has effect. A list of the option
keywords, along with their abbreviations, defaults, and meanings
follows:

KEYWORD	ABBREVIATION	DEFAULT VALUE
SOURCE	S	Batch: SOURCE
NOSOURCE	NS	Terminal: NOSOURCE

Requests that a listing of the MAD/I source program be written to SPRINT.

DECK	D	DECK
NODECK	ND	

Requests that the generated object module be written to SPUNCH.

LIST	L	NOLIST
NOLIST	NL	

Requests that a listing of the generated machine instructions and a storage map be written to SPRINT.

MAP	M	NOMAP
NOMAP	NM	

Requests that a storage map showing the storage assignments of all variables and constants be written to SPRINT.

XREF	X	NOXREF
NOXREF	NX	

Requests that a cross-reference table for all the identifiers in the program be written to SPRINT.

ATR	A	Batch: ATR
NOATR	NA	Terminal: NOATR

Requests that a list of the attributes of each identifier be written to SPRINT.

OPLIST	OL	Batch: OPLIST
NOOPLIST	NOL	Terminal: NOOPLIST

Requests that a listing of the option assignments for this compilation be written to SPRINT.

```
SORMGIN=(m,n)        SM=              SORMGIN=(1,256)
    =m,n
    =(m  n)
    =m  n
```

Specifies the left and right margins of the source
program lines to be m and n, respectively, where 1≤m<n≤256.
All text outside of this range is ignored.   For instance,
to read source lines which have sequence-id information in
columns 73 to 80, specify m=1 and n=72.

```
FREEFORM             FF               FREEFORM
LINEFORM  .          LF
```

FREEFORM specifies that the input text is completely
free-form, extending from line to line as a continuous
sequence of characters, with statements separated   by
semicolons.   LINEFORM specifies that each input line will
have a semicolon automatically appended to it unless the
last character (the one at the right margin) is the
continuation character.   The continuation character is
specified with CCNTCHAR option.

```
CONTCHAR=c           CC=              CONTCHAR=+
```

Specifies that c is the continuation character to be
used in conjunction with the LINEFORM option.

```
SOURCETAB=n          ST=              SOURCETAB=6
```

Specifies that the source program, if it is printed,
be printed beginning in column n.   The source program
listing itself is controlled by the SOURCE option.

```
SIZE=(m,n)                            SIZE=(3,255)
    =m,n
    =(m  n)
    =m  n
```

Specifies the sizes of two internal translator tables.
M specifies the maximum number of control sections. N
specifies the maximum number of "basetab" entries.   These
need not be given except for very large programs.

Section 12: Sample Runs of MAD/I in MTS

12.1 Sample Run of CALLSQRT and SQRT

The following excerpt from a terminal session shows the
runs of the MAD/I compiler used to generate the sample output of
Section 10.1. Notice that the compiler is run twice, once for
each program. Also notice that the defaults for terminal
operation are such that no listings are produced. In this and
all following examples lower-case characters are typed by the
user. Lines preceded by a "#" are commands to MTS. Some lines
have been truncated on the right to fit within the column width
of this report.

```
#list callsqrt
>        1                'PROCEDURE' CALLSQRT.;
>        2     CALLSQRT: 'WRITE'("'&ENTER X:'*");
>        3                'READ' ("WF*"), X;
>        4                'WRITE' (" ' X=',WF,' SQRT OF X=',WF*"),X,
>        5                'GO TO' CALLSQRT
>        6                'END'
>        7
>        8
>        9
>       10
>       11                'PROCEDURE' SQRT.(X);
>       12                'PRESET' EPS := .0001;
>       13     SQRT:      'IF' X=0. | X=1., 'RETURN' X;
>       14                Y := X;
>       15     LOOP:      Z := (Y+X/Y)/2.;
>       16                'IF' .ABS.(Y-Z) < EPS, 'RETURN' Z;
>       17                Y := Z;
>       18                'GO TO' LOOP
>       19                'END'
#END OF FILE
#run *mad1 scards=callsqrt(1,10) spunch=-load
#EXECUTION BEGINS

        MAD/I COMPILER VERSION PR240-093943.

   MAD/I COMPILER STATISTIC      PASS1        ALLOC        PASS2
   CPU TIME              (SEC)   1.125        1.575        2.981
   ELAPSED TIME         (SEC)    2.423        2.637        8.093
   CPU VM INTEGRAL (PG-SEC)    168.693      238.740      456.407      8
   MEAN VM SIZE        (PGS)    113.569      116.595      121.376      3
   DRUM READS                    29           27
   STATEMENTS                     5
   DESCRIPTORS                   35
#EXECUTION TERMINATED
#run *mad1 scards=callsqrt(11) spunch=-load(last+1)
```

#EXECUTION BEGINS

        MAD/I COMPILER VERSION PR240-093943.

    MAD/I COMPILER STATISTIC   PASS1   ALLOC   PASS2
    CPU TIME   (SEC)   .988   1.464   2.854
    ELAPSED TIME   (SEC)   1.343   1.654   3.496
    CPU VM INTEGRAL (PG-SEC)   147.653   222.130   436.374   8
    MEAN VM SIZE   (PGS)   125.010   125.994   127.794   3
    DRUM READS   1   4
    STATEMENTS   10
    DESCRIPTORS   65
#EXECUTION TERMINATED
#run -load
#EXECUTION BEGINS
    ENTER X: 100.0
    X=   100.0000 SQRT OF X=   10.0000
    ENTER X: 1.0
    X=   1.0000 SQRT OF X=   1.0000
    ENTER X: 0.
    X=   .0000 SQRT OF X=   .0000
    ENTER X: 4.0
    X=   4.0000 SQRT OF X=   2.0000
    ENTER X: ¢

    **** ALL INPUT DATA HAS BEEN PROCESSED - AT LOCATION 500788
#EXECUTION TERMINATED

12.2 Sample Run of HASHTEST and HASH

    The following excerpt from a terminal session shows the runs of the MAD/I compiler used to generate the sample output of Section 10.2. The option ol chosen on the first run caused all the compiler option assignments to be printed. Likewise, the source option on both compilations caused the source listings to be produced. Note that on line 14 of HASH the 'NULL PT' has been replaced by 0. This is necessary due to a minor bug in the compiler which does not allow 'NULL PT' to work properly in a 'PRESET' statement.

```
#empty -deck
#DONE.
#run *mad1 scards=hashtest spunch=-deck par=source,ol
 EXECUTION BEGINS
```

MAD/I COMPILER OPTION ASSIGNMENTS:

  SOURCE,DECK,NOLIST,SORMGIN=(001,256),FREEFORM,CONTCHAR
  SOURCETAB=006,SIZE=(0003,0255),COMPILE
  NOMAP,NOXREF,NOATR,OPLIST,USER,ADDENDA

MAD/I COMPILER VERSION PR240-093943.

MAD/I COMPILER SOURCE PROGRAM LISTING ... ...

```
0001              'PROCEDURE' HASHTEST.;
0002              'DECLARE' 'DEFAULT' 'INTEGER';
0003              'DECLARE' HASH 'ENTRY POINT' 'POINTER';
0004              'DECLARE' PTR 'POINTER';
0005              'DECLARE' SYMENT 'BASED' 'COMPONENT STRUCTURE'(
0006                'POINTER',
0007                'CHARACTER'(8),
0008                'INTEGER',
0009                'BIT'(8) );
0010              'DECLARE' SYMBOL 'CHARACTER'(8);
0011 HASHTEST:    'WRITE'("'&ENTER NEXT SYMBOL:'*");
0012              'READ'("C8.8*"), SYMBOL;
0013              PTR := HASH.(SYMBOL);
0014              SYMENT .ALLOC. PTR;
0015              'WRITE'("'SYMBOL TABLE ENTRY AT: ',X8.4,' PTR='
0016                X8.4,' NAME=',C8.8*"), PTR,
0017                SYMENT(1),SYMENT(2);
0018              'GO TO' HASHTEST;
0019              'END'
```

MAD/I COMPILER STATISTIC		PASS1	ALLOC	PASS2	
CPU TIME	(SEC)	2.151	2.307	3.612	
ELAPSED TIME	(SEC)	3.370	3.364	4.930	
CPU VM INTEGRAL	(PG-SEC)	322.617	350.213	552.274	12
MEAN VM SIZE	(PGS)	88.212	88.768	89.635	2
DRUM READS		13	17	4	
STATEMENTS		13			
DESCRIPTORS		92			

#EXECUTION TERMINATED
#run *mad1 scards=hash spunch=-deck(1000) par=s
#EXECUTION BEGINS

    MAD/I COMPILER VERSION PR240-093943.

    MAD/I COMPILER SOURCE PROGRAM LISTING ... ...

```
0001              'PROCEDURE' HASH.(SYMBOL);
0002
0003              'DECLARE DEFAULT' 'INTEGER';
0004              'DECLARE' SYMBOL 'CHARACTER'(8);
```

Part II -- User's Guide for MAD/I in MTS

```
0005                'DECLARE' HASH 'ENTRY POINT' 'POINTER';
0006                'DECLARE' SYMENT 'BASED' 'COMPONENT STRUCTURE'(
0007                 'POINTER',              << NEXT SYMBOL >>
0008                 'CHARACTER'(8),    << SYMBOL NAME >>
0009                 'INTEGER',              << STORAGE ALLOC >>
0010                 'BIT'(8) );            << CLASS MODE >>
0011                'DECLARE' THREADS 'FIXED ARRAY'(0...6) 'POINTER
0012                'DECLARE' FINGER 'POINTER';
0013                'DECLARE' NAMES 'FIXED ARRAY'(2) 'INTEGER';
0014                'PRESET' THREADS := 7(0);
0015
0016  HASH:        (NAMES .AS.('CHARACTER'(8))) := SYMBOL;
0017                KEY := .ABS.((NAMES(1)+NAMES(2)).REM.7);
0018                'WRITE' ("'**** KEY=',I*"),KEY;
0019                FINGER := THREADS(KEY);
0020  LOOP:        'IF' FINGER = 'NULL PT';
0021                 'ALLOCATE' SYMENT;
0022               . SYMENT(1) := THREADS(KEY);
0023               ˉ THREADS(KEY) := .PT. SYMENT;
0024                 SYMENT(2) := SYMBOL;
0025                 SYMENT(4) := SYMENT(3) := 0;
0026               'ELSE';
0027                 SYMENT .ALLOC. FINGER;
0028                 'IF' SYMBOL = SYMENT(2), 'GO TO' FOUND;
0029                 FINGER := SYMENT(1);
0030                 'GO TO' LOOP
0031                'END';
0032  FOUND:       'RETURN' .PT. SYMENT
0033                'END'
```

```
MAD/I COMPILER STATISTIC     PASS1        ALLOC        PASS2
CPU TIME              (SEC)     2.928       3.026        7.363
ELAPSED TIME         (SEC)     4.663       4.773       11.367
CPU VM INTEGRAL (PG-SEC)     439.160     459.257     1126.523    20
MEAN VM SIZE         (PGS)     92.347      92.970       94.465     2
DRUM READS                       29          18           11
STATEMENTS                       26
DESCRIPTORS                     202                                 2
#EXECUTION TERMINATED
#run -deck
#EXECUTION BEGINS
  ENTER NEXT SYMBOL:  a
 **** KEY=        1
 SYMBOL TABLE ENTRY AT: 00500068 PTR=00000000 NAME=A
  ENTER NEXT SYMBOL:  b
 **** KEY=        2
 SYMBOL TABLE ENTRY AT: 00500080 PTR=00000000 NAME=B
  ENTER NEXT SYMBOL:  c
 **** KEY=        3
 SYMBOL TABLE ENTRY AT: 00500098 PTR=00000000 NAME=C
  ENTER NEXT SYMBOL:  d
 **** KEY=        4
```

```
SYMBOL TABLE ENTRY AT: 005000B0 PTR=00000000 NAME=D
 ENTER NEXT SYMBOL:  e
**** KEY=       5
SYMBOL TABLE ENTRY AT: 00500C68 PTR=00000000 NAME=E
 ENTER NEXT SYMBOL:  f
**** KEY=       6
SYMBOL TABLE ENTRY AT: 00500C80 PTR=00000000 NAME=F
 ENTER NEXT SYMBOL:  g
**** KEY=       0
SYMBOL TABLE ENTRY AT: 00500C98 PTR=00000000 NAME=G
 ENTER NEXT SYMBOL:  h
**** KEY=       1
SYMBOL TABLE ENTRY AT: 00500CB0 PTR=00500068 NAME=H
 ENTER NEXT SYMBOL:  i
**** KEY=       2
SYMBOL TABLE ENTRY AT: 00500CC8 PTR=00500080 NAME=I
 ENTER NEXT SYMBOL:  a
**** KEY=       1
SYMBOL TABLE ENTRY AT: 00500068 PTR=00000000 NAME=A
 ENTER NEXT SYMBOL:  h
**** KEY=       1
SYMBOL TABLE ENTRY AT: 00500CB0 PTR=00500068 NAME=H
 ENTER NEXT SYMBOL:  aardvark
**** KEY=       4
SYMBOL TABLE ENTRY AT: 00500CE0 PTR=005000B0 NAME=AARDVARK
 ENTER NEXT SYMBOL:  quail
**** KEY=       0
SYMBOL TABLE ENTRY AT: 00500CF8 PTR=00500C98 NAME=QUAIL
 ENTER NEXT SYMBOL:  wunerful
**** KEY=       2
SYMBOL TABLE ENTRY AT: 00500D10 PTR=00500CC8 NAME=WUNERFUL
 ENTER NEXT SYMBOL:  a
**** KEY=       1
SYMBOL TABLE ENTRY AT: 00500068 PTR=00000000 NAME=A
 ENTER NEXT SYMBOL:  ¢

    **** ALL INPUT DATA HAS BEEN PROCESSED - AT LOCATION 5009E0
#EXECUTION TERMINATED
```

12.3   Sample Run of Combined CALLSORT and SORT

The following excerpt from a terminal session shows a run
of the MAD/I compiler on the procedures CALLSORT and SORT as
combined into one program. All compiler output (except for
internal compiler debugging aids) is turned on in this example.
The output is described in some detail below.

The first page consists of the option assignments and
source program listing. Each line of the program is given a
line number which is used as a reference in error messages and
object program listing.

The next page gives the storage allocation of the constants
in the program. Other constants are generated as needed and are
printed interspersed with the object program listing which
follows. The two-byte and six-byte fields at the beginning of
each line are the control section identification and relocatable
address (within the control section) of the beginning of the
data. The third field is the text of the constant. All numbers
are in hexadecimal.

The next five pages are a listing of the generated object
program. The object code is preceded by the line or lines which
caused it to be generated. The first three fields are the
control section identification, relocatable address, and text,
as described above. A "+" is printed in lines which set out-of-
line text. There are two types of out-of-line text. First,
instructions which reference addresses not yet generated are
modified (actually, completed) by out-of-line text when the
forward reference is resolved. Second, additional constants and
internal variables are allocated out-of-line as required. The
remainder of the line is a pseudo-assembler code representation
of the line. Run-time symbol table entries and the base table
(used for addressability) are generated at the end of the object
program listing.

Next come two pages giving the external symbol dictionary
and relocation dictionary. The notation used in these tables is
similar to that used in other System/360 translators.

The next page of output gives a storage map showing the
allocation of all variables and constants in the program. The
first field gives the control section identification of the
allocation. If the item has no allocation in this program, "00"
is given as the control section identification. The next field
gives the storage class of the item. The correspondence is as
follows:

        01   Static
        02   External

```
03          Formal Parameter
07          Based
```

The next field gives the displacement within the base table of
the base address constant to be used in referencing this item.
Notice that formal parameters, external symbols, and based
variables always have a unique base table entry, while many
static items may be referenced using the same base table entry.
The last field gives the relocatable address of the item within
the control section.

The last page gives the attributes of each symbol in the
program and is self-explanatory.   The numeric fields are
identical to those given in the storage map.


```
#empty -deck
#DONE.
#run *mad1 scards=callsqrt2 spunch=-deck par=s,a,l,m,ol
#EXECUTION BEGINS
```

MAD/I COMPILER OPTION ASSIGNMENTS:

    SOURCE,DECK,LIST,SORMGIN=(001,256),FREEFORM,CONTCHAR=+
    SOURCETAB=006,SIZE=(0003,0255),COMPILE
    MAP,NOXREF,ATR,OPLIST,USER,ADDENDA

MAD/I COMPILER VERSION PR240-093943.

MAD/I COMPILER SOURCE PROGRAM LISTING ... ...

```
0001              'PROCEDURE' CALLSQRT.;
0002 CALLSQRT: 'WRITE'("'&ENTER X:'*");
0003              'READ' ("WF*"), X;
0004              'WRITE' ("' X=',WF,' SQRT OF X=',WF*"),X,SQRT.(
0005              'GO TO' CALLSQRT;
0006
0007              'PROCEDURE' SQRT.(X);
0008              'PRESET' EPS := .0001;
0009 SQRT:        'IF' X=0. | X=1., 'RETURN' X;
0010              Y := X;
0011 LOOP:        Z := (Y+X/Y)/2.;
0012              'IF' .ABS.(Y-Z) < EPS, 'RETURN' Z;
0013              Y := Z;
0014              'GO IC' LOOP
0015              'END'
0016              'END'
```

STORAGE ALLOCATION

```
01 000000                          #CALLSQR CSECT
01 000064 41200000              +          CONST 2.
01 000068 41100000              +          CONST 1.
01 00006C 00000000              +          CONST 0.
01 000070 7D40E77E7D6BE6C66B7D +          CONST "' X=',WF,' SQ
01 00008A E6C65C               +          CONST "WF*"
01 00008D 7D50C5D5E3C5D940E77A +          CONST "'&ENTER X:'*"
```

MAD/I COMPILER OBJECT PROGRAM LISTING ... ...

```
 02 000000                               @CALLSQR CSECT
*0001                    'PROCEDURE' CALLSQRT.;
*0002    CALLSQRT: 'WRITE'("'&ENTER X:'*");
 02 000000                               CNOP   0,4
 02 000000                      CALLSQRT EQU    *
 02 000000  90ECD00C                     STM    14,12,12(13)
 02 000004  58C0F020                     L      12,32(,15)
 02 000008  58E0C00C                     L      14,%STKADR
 02 00000C  58E0E004                     L      14,4(,14)
 02 000010  50E0D008                     ST     14,8(,13)
 02 000014  50D0E004                     ST     13,4(,14)
 02 000018  18DE                         LR     13,14
 02 00001A  47F0F028                     B      40(,15)
 02 000020  00000000                     DC     A(%BASETAB)
 02 000024  58C0F020                     L      12,32(,15)
 02 000028  1B11                         SR     1,1
 02 00002A  5840C00C                     L      4,%STKADR
 02 00002E  41E0D048                     LA     14,72(,13)
 02 000032  98234000                     LM     2,3,0(4)
 02 000036  90DE4000                     STM    13,14,0(4)
 02 00003A  58F0C014                     L      15,#+20
 02 00003E  0DEF                         BASR   14,15
 02 000040  90234000                     STM    2,3,0(4)
 02 000044  58B0C000                     L      11,0(,12)
 02 000048  50F0B09C                     ST     15,%RTNCODE
 01 00005C  0000008D          +          DC     A("'&ENTER X:'
 01 000060  00300C00          +          CONST  3148800
 01 000058  00000001          +          CONST  1
 02 00004C  4110B05C                     LA     1,#CALLSQR+92
 02 000050  5840C00C                     L      4,%STKADR
 02 000054  41E0D048                     LA     14,72(,13)
 02 000058  98234000                     LM     2,3,0(4)
 02 00005C  90DE4000                     STM    13,14,0(4)
 02 000060  58F0C018                     L      15,#+24
 02 000064  0DEF                         BASR   14,15
 02 000066  90234000                     STM    2,3,0(4)
 02 00006A  50F0B09C                     ST     15,%RTNCODE
 02 00006E  1B11                         SR     1,1
 02 000070  5840C00C                     L      4,%STKADR
 02 000074  41E0D048                     LA     14,72(,13)
 02 000078  98234000                     LM     2,3,0(4)
 02 00007C  90DE4000                     STM    13,14,0(4)
 02 000080  58F0C01C                     L      15,#+28
 02 000084  0DEF                         BASR   14,15
 02 000086  90234000                     STM    2,3,0(4)
 02 00008A  50F0B09C                     ST     15,%RTNCODE
*0003                    'READ' ("WF*"), X;
 02 00008E  1B11                         SR     1,1
 02 000090  5840C00C                     L      4,%STKADR
 02 000094  41E0D048                     LA     14,72(,13)
```

```
02 000098 98234000                        LM    2,3,0(4)
02 00009C 90DE4000                        STM   13,14,0(4)
02 0000A0 58F0C020                        L     15,#+32
02 0000A4 0DEF                            BASR  14,15
02 0000A6 90234000                        STM   2,3,0(4)
02 0000AA 50F0B09C                        ST    15,%RTNCODE
01 000050 0000008A              +         DC    A("WF*")
01 000054 00300300              +         CONST 3146496
01 00004C 00000001              +         CONST 1
02 0000AE 4110B050                        LA    1,#CALLSQR+80
02 0000B2 5840C00C                        L     4,%STKADR
02 0000B6 41E0D048                        LA    14,72(,13)
02 0000BA 98234000                        LM    2,3,0(4)
02 0000BE 90DE4000                        STM   13,14,0(4)
02 0000C2 58F0C018                        L     15,#+24
02 0000C6 0DEF                            BASR  14,15
02 0000C8 90234000                        STM   2,3,0(4)
02 0000CC 50F0B09C                        ST    15,%RTNCODE
01 000044 0000000C              +         DC    A(X)
01 000048 00720400              +         CONST 7472128
01 000040 00000001              +         CONST 1
02 0000D0 4110B044                        LA    1,#CALLSQR+68
02 0000D4 5840C00C                        L     4,%STKADR
02 0000D8 41E0D048                        LA    14,72(,13)
02 0000DC 98234000                        LM    2,3,0(4)
02 0000E0 90DE4000                        STM   13,14,0(4)
02 0000E4 58F0C024                        L     15,#+36
02 0000E8 0DEF                            BASR  14,15
02 0000EA 90234000                        STM   2,3,0(4)
02 0000EE 50F0B09C                        ST    15,%RTNCODE
02 0000F2 1B11                            SR    1,1
02 0000F4 5840C00C                        L     4,%STKADR
02 0000F8 41E0D048                        LA    14,72(,13)
02 0000FC 98234000                        LM    2,3,0(4)
02 000100 90DE40C0                        STM   13,14,0(4)
02 000104 58F0C01C                        L     15,#+28
02 000108 0DEF                            BASR  14,15
02 00010A 90234000                        STM   2,3,0(4)
02 00010E 50F0B09C                        ST    15,%RTNCODE
*0004                  'WRITE' (" X=',WF,' SQRT OF X=',WF*"),X,SQR
02 000112 1B11                            SR    1,1
02 000114 5840C00C                        L     4,%STKADR
02 000118 41E0D048                        LA    14,72(,13)
02 00011C 98234000                        LM    2,3,0(4)
02 000120 90DE4000                        STM   13,14,0(4)
02 000124 58F0C014                        L     15,#+20
02 000128 0DEF                            BASR  14,15
02 00012A 90234000                        STM   2,3,0(4)
02 00012E 50F0B09C                        ST    15,%RTNCODE
01 000038 00000070              +         DC    A("' X=',WF,'
01 00003C 00301A00              +         CONST 3152384
01 000034 00000001              +         CONST 1
```

Part II -- User's Guide for MAD/I in MTS

```
02 000132 4110B038              LA    1,#CALLSQR+56
02 000136 5840C00C              L     4,%STKADR
02 00013A 41E0D048              LA    14,72(,13)
02 00013E 98234000              LM    2,3,0(4)
02 000142 90DE4000              STM   13,14,0(4)
02 000146 58F0C018              L     15,#+24
02 00014A 0DEF                  BASR  14,15
02 00014C 90234000              STM   2,3,0(4)
02 000150 50F0B09C              ST    15,%RTNCODE
01 00002C 0000000C       +      DC    A(X)
01 000030 00720400       +      CONST 7472128
01 000028 00000001       +      CONST 1
02 000154 4110B02C              LA    1,#CALLSQR+44
02 000158 5840C00C              L     4,%STKADR
02 00015C 41E0D048              LA    14,72(,13)
02 000160 98234000              LM    2,3,0(4)
02 000164 90DE4000              STM   13,14,0(4)
02 000168 58F0C024              L     15,#+36
02 00016C 0DEF                  BASR  14,15
02 00016E 90234000              STM   2,3,0(4)
02 000172 50F0B09C              ST    15,%RTNCODE
01 000020 0000000C       +      DC    A(X)
01 000024 00720400       +      CONST 7472128
01 00001C 00000001       +      CONST 1
02 000176 4110B020              LA    1,#CALLSQR+32
02 00017A 5840C00C              L     4,%STKADR
02 00017E 41E0D048              LA    14,72(,13)
02 000182 98234000              LM    2,3,0(4)
02 000186 90DE4000              STM   13,14,0(4)
02 00018A 58A0C000              L     10,0(,12)
02 00018E 41F0A000              LA    15,SQRT
02 000192 0DEF                  BASR  14,15
02 000194 90234000              STM   2,3,0(4)
02 000198 50F0B09C              ST    15,%RTNCODE
02 00019C 7000B0A0              STE   0,%TMP0001
01 000014 000000A0       +      DC    A(%TMP0001)
01 000018 00720400       +      CONST 7472128
01 000010 00000001       +      CONST 1
02 0001A0 4110B014              LA    1,#CALLSQR+20
02 0001A4 5840C00C              L     4,%STKADR
02 0001A8 41E0D048              LA    14,72(,13)
02 0001AC 98234000              LM    2,3,0(4)
02 0001B0 90DE4000              STM   13,14,0(4)
02 0001B4 58F0C024              L     15,#+36
02 0001B8 0DEF                  BASR  14,15
02 0001BA 90234000              STM   2,3,0(4)
02 0001BE 50F0B09C              ST    15,%RTNCODE
02 0001C2 1B11                  SR    1,1
02 0001C4 5840C00C              L     4,%STKADR
02 0001C8 41E0D048              LA    14,72(,13)
02 0001CC 98234000              LM    2,3,0(4)
02 0001D0 90DE4000              STM   13,14,0(4)
```

```
02 0001D4  58F0C01C                            L     15,#+28
02 0001D8  0DEF                                BASR  14,15
02 0001DA  90234000                            STM   2,3,0(4)
02 0001DE  50F0B09C                            ST    15,%RTNCODE
*0005                    'GO TO' CALLSQRT;
02 0001E2  1B11                                SR    1,1
02 000028                            0         EQU   CALLSQRT+40
02 00C1E4  5890C008                            L     9,8(,12)
02 0001E8  47F09028                            B     9CALLSQR+40
*0006
*0007                    'PROCEDURE' SQRT.(X);
*0008                    'PRESET' EPS := .0001;
01 000000  3A2AF31D              +             CONST .0001
*0009   SQRT:        'IF' X=0. | X=1., 'RETURN' X;
02 0001EC                                      CNOP  0,4
02 0001EC                          SQRT        EQU   *
02 00018C  C008                      +
02 000190  A1EC                      +
02 0001EC  90ECD00C                            STM   14,12,12(13)
02 0001F0  58C0F020                            L     12,32(,15)
02 0001F4  58E0C00C                            L     14,%STKADR
02 0001F8  58E0E004                            L     14,4(,14)
02 0001FC  50E0D008                            ST    14,8(,13)
02 000200  50D0E004                            ST    13,4(,14)
02 000204  18DE                                LR    13,14
02 000206  47F0F028                            B     40(,15)
02 00020C  00000000                            DC    A(%BASETAB)
02 000210  58C0F020                            L     12,32(,15)
02 000214  58201000                            L     2,0(,1)
02 000218  5020C010                            ST    2,#+16
02 00021C  58B0C010                            L     11,16(,12)
02 000220  7820B000                            LE    2,X
02 000224  58A0C000                            L     10,0(,12)
02 000228  7920A06C                            CE    2,=0.
02 00022C  92FFA0A4                            MVI   %TMP0001,"FF"X
02 000230  5890C000                            L     9,0(,12)
02 000234  47809000                            BE    %FLA0002
02 000238  9200A0A4                            MVI   %TMP0001,0
02 00023C                          %FLA0002    EQU   *
02 000232  C008                      +
02 000236  923C                      +
02 00023C  7920A068                            CE    2,=1.
02 000240  92FFA0A6                            MVI   %TMP0002,"FF"X
02 000244  5880C000                            L     8,0(,12)
02 000248  47808000                            BE    %FLA0003
02 00024C  9200A0A6                            MVI   %TMP0002,0
02 000250                          %FLA0003    EQU   *
02 000246  C008                      +
02 00024A  8250                      +
02 000250  D200A0A5A0A4                         MVC   %TMP0003(1),%T
02 000256  D600A0A5A0A6                         OC    %TMP0003(1),%T
02 00025C  9500A0A5                            CLI   %TMP0003,0
```

```
02 000260 5870C000                              L      7,0(,12)
02 000264 47807000                              BE     %FLD0005
02 000268 3802                                  LER    0,2
02 00026A 58D0D004                              L      13,4(,13)
02 00026E 98ECD00C                              LM     14,12,12(13)
02 000272 1BFF                                  SR     15,15
02 000274 07FE                                  BR     14
02 000276                          %FLD0005 EQU *
02 000262 C008                              ♦
02 000266 7276                              ♦
*0010                Y := X;
02 000276 58B0C000                              L      11,0(,12)
02 00027A 58A0C010                              L      10,16(,12)
02 00027E D203B004A000                          MVC    Y(4),X
*0011   LOOP:        Z := (Y+X/Y)/2.;
02 000284                          LOOP     EQU *
02 000284 58B0C010                              L      11,16(,12)
02 000288 7820B000                              LE     2,X
02 00028C 58A0C000                              L      10,0(,12)
02 000290 7D20A004                              DE     2,Y
02 000294 7A20A004                              AE     2,Y
02 000298 7D20A064                              DE     2,=2.
02 00029C 7020A008                              STE    2,Z
*0012                'IF' .ABS.(Y-Z) < EPS, 'RETURN' Z;
02 0002A0 7820A004                              LE     2,Y
02 0002A4 7B20A008                              SE     2,Z
02 0002A8 3022                                  LPER   2,2
02 0002AA 7920A000                              CE     2,EPS
02 0002AE 5890C000                              L      9,0(,12)
02 0002B2 47B09000                              BNL    %FLD0007
02 0002B6 7800A008                              LE     0,Z
02 0002BA 58D0D004                              L      13,4(,13)
02 0002BE 98ECD00C                              LM     14,12,12(13)
02 0002C2 1BFF                                  SR     15,15
02 0002C4 07FE                                  BR     14
02 0002C6                          %FLD0007 EQU *
02 0002B0 C008                              ♦
02 0002B4 92C6                              ♦
*0013                Y := Z;
02 0002C6 58B0C000                              L      11,0(,12)
02 0002CA D203B004B008                          MVC    Y(4),Z
*0014                'GO TO' LOOP
*0015                'END'
02 0002D0 58A0C008                              L      10,8(,12)
02 0002D4 47F0A284                              B      LOOP
*0016                'END'
*0016    @ICODEENDOFFILE
```

RTST ENTRIES FOR BLOCK %BLN0001

RTST ENTRIES FOR BLOCK %BLN0002

```
01 0000A8                              %BASETAB EQU     %BASETAB
02 00020C 000000A8                     +
02 000020 000000A8                     +
01 0000A8 00000000                     +
01 0000AC 000000A8                     +
01 0000B0 00000000                     +
01 0000B4 00000000                     +
01 0000B8 00000000                     +
01 0000BC 00000000                     +
01 0000C0 00000000                     +
01 0000C4 00000000                     +
01 0000C8 00000000                     +
01 0000CC 00000000                     +
```

EXTERNAL SYMBOL DICTIONARY (SYMBOL,TYPE,ID,ADDR,LENGTH/LDID)

```
#CALLSQR PD 01 000000 0000D0
@CALLSQR SD 02 000000 0002D8
MADSTACK ER 03
CALLSQRT LD    000000 000002
MADWRITE ER 04
FORMAT   ER 05
ENDIOP   ER 06
MACREAD  ER 07
IOP      ER 08
```

RELOCATION DICTIONARY   (P.ID,R.ID,FLAGS,ADDRESS)

```
01  01  0C  00005C
01  01  0C  000050
01  01  0C  000044
01  01  0C  000038
01  01  0C  00002C
01  01  0C  000020
01  01  0C  000014
02  01  0C  00020C
02  01  0C  000020
01  01  0C  0000A8
01  01  0C  0000AC
01  02  0C  0000B0
01  03  0C  0000B4
01  04  0C  0000BC
01  05  0C  0000C0
01  06  0C  0000C4
01  07  0C  0000C8
01  08  0C  0000CC
02 000000                                END    CALLSQRT
```

STORAGE MAP

```
00 02 0014 000004 MADWRITE
00 02 0018 000005 FORMAT
00 02 001C 000006 ENDIOP
00 02 0020 000007 MADREAD
00 02 0024 000008 IOP
00 03 0010 000000 X
01 01 0000 000000 EPS
01 01 0000 000004 Y
01 01 0000 000008 Z
01 01 0000 00000C X
01 01 0000 000064 2.
01 01 0000 000068 1.
01 01 0000 00006C 0.
01 01 0000 000070 "' X=',WF,' SQRT OF X=',WF*"
01 01 0000 00008A "WF*"
01 01 0000 00008D "'&ENTER X:'*"
01 01 0000 00009C %FINCODE
01 01 0000 0000A8 %EASETAB
02 01 0008 000000 CALLSQRT
02 01 0008 0001EC SQRT
02 01 0008 000284 LOOP
```

SYMBOL ATTRIBUTES

   BLOCK %BLN0001   NUMBER OF SYMBOLS=19

'DEFAULT' 'FLOATINGSHORT' 00 00 0000 000000
%RTNCODE 'INTEGERLONG' 01 01 0000 C0009C
CALLSQRT 'ENTRYPOINT' 02 01 0008 000000 'ACCESSIBLE'
   RESULT= 'FLOATINGSHORT'
ENDIOP 'ENTRYPOINT' 00 02 001C 000006 'EXTERNAL'
EPS 'FLOATINGSHORT' 01 01 0000 000000
FORMAT 'ENTRYPOINT' 00 02 0018 000005 'EXTERNAL'
IOP 'ENTRYPCINT' 00 02 0024 000008 'EXTERNAL'
MADREAD 'ENTRYPOINT' 00 02 0020 000007 'EXTERNAL'
MADWRITE 'ENTRYPOINT' 00 02 0014 000004 'EXTERNAL'
SQRT 'ENTRYPOINT' 02 01 0008 0001EC
   RESULT= 'FLOATINGSHORT'
X 'FLOATINGSHORT' 01 01 0000 00000C
Y 'FLOATINGSHORT' 01 01 0000 000004
Z 'FLOATINGSHORT' 01 01 0000 000008
"' X=',WF,' SQRT OF X=',WF*" 'CHARACTER' 01 01 0000 000070
   LENGTH=26
"'&ENTER X:'*" 'CHARACTER' 01 01 0000 00008D
   LENGTH=12
"WF*" 'CHARACTER' 01 01 0000 00008A
   LENGTH=3
0. 'FLOATINGSHORT' 01 01 0000 00006C
1. 'FLOATINGSHORT' 01 01 0000 000068
2. 'FLOATINGSHORT' 01 01 0000 000064

   BLOCK %BLN0002   NUMBER OF SYMBOLS=2

LOOP 'TRANSFERPOINT' 02 01 0008 000284
X 'FLOATINGSHORT' 00 03 0010 000000 (FORMAL PAR)

MAD/I COMPILER STATISTIC		PASS1	ALLOC	PASS2	
CPU TIME	(SEC)	2.115	2.649	10.213	
ELAPSED TIME	(SEC)	6.067	6.946	29.114	
CPU VM INTEGRAL	(PG-SEC)	316.617	402.356	1565.634	22
MEAN VM SIZE	(PGS)	79.199	79.600	81.137	2
DRUM READS		57	77	245	3
STATEMENTS		15			
DESCRIPTORS		100			1

#EXECUTION TERMINATED
#run -deck map

... ...

 ENTRY = 5001A8   SIZE = 00802D

NAME	VALUE	T RF	NAME	VALUE	T RF	NAME	VALUE

```
GETSPACE 20DD9E *      FREESPAC 20E09E *      LOAD     20F7B0
SYSTEM   2157CC *      ERROR    2157F6 *      PGNTTRP  2181CC
GETFD    218878 *      SCARDS   218B34 *      SPRINT   218B46
SPUNCH   218B58 *      SERCOM   218B6A *      READ     218BE8
WRITE    218C04 *      LCSYMBOL 2197D0 *      #CALLSQR 5000D8
@CALLSQR 5001A8  5001A8 SPIE    500480 *500480 MADIO   5005F0
MADREAD  5005F0 *      MADWRITE 50061E *      FORMAT   50073A
IOP      50077C *      ENDIOP   5007C2 *      MDIOPSCT 500958
MADSTACK 503000 *503000 IOH360  504000 *504000 IOHIN   5040F0
IOHOUT   504114 *      IOHETC   50483C *      ONE@ATIM 50492C
IOHERP   508000 *503F18 GLAP    50A000 *506988 IOPKG   50B000
ROPEN    50B0CE *      RCLOSE   50B148 *      POPEN    50B174
PCLOSE   50B1C0 *
```

    ... ...

```
#EXECUTION BEGINS
  ENTER X:   100.0
  X=     100.0000 SQRT OF X=      10.0000
  ENTER X:   1.0
  X=       1.0000 SQRT OF X=       1.0000
  ENTER X:   0.
  X=        .0000 SQRT OF X=        .0000
  ENTER X:   2.0
  X=       2.0000 SQRT OF X=       1.4142
  ENTER X:   4.0
  X=       4.0000 SQRT OF X=       2.0000
  ENTER X:   ¢
 0**** ALL INPUT DATA HAS BEEN PROCESSED - AT LOCATION 500768
#EXECUTION TERMINATED
```

Section 13: MAD/I Error Messages

This section has not yet been written -- sorry.

Section 14: Object Module Description

14.1 Representation of Data

Alignment Attribute

The alignment attribute of an item specifies a constraint on the positioning of its allocated storage. The alignment attribute for an item is taken as the maximum of the value explicitly declared through the 'ALIGN' keyword (if any) and the alignment implied by other attributes of the item. The valid alignment values and their definitions are:

        1: Any byte boundary.
        2: Any halfword boundary.
        4: Any fullword boundary.
        8: Any double-word boundary.

Mode Representations

The following table gives the internal representations used for the various MAD/I modes. Representation terminology is defined in the IBM System/360 Principles of Operation manual. The "length" given is the length in bytes.

Mode	Alignment	Length	Representation
'INTEGER SHORT'	2	2	Halfword fixed-point number.
'INTEGER LONG'	4	4	Fullword fixed-point number.
'FLOATING SHORT'	4	4	Short floating-point number.
'FLOATING LONG'	8	8	Long floating-point number.
'PACKED' (n)	1	n	Packed-decimal number.
'BIT' (n)	-	-	n bits, allocated such that all bits are contained in one fullword.
'BOOLEAN'	1	1	A logical byte; all bits 1 represents 'TRUE' and all bits 0 represents 'FALSE'.
'CHARACTER' (n)	1	n	Variable-length logical information; i.e., n bytes

			representing n characters in EBCDIC.
'VARYING CHARACTER' (n)	2	n+2	The halfword fixed-point number representing the current length of the character string, followed by the characters, one per byte.
'FILE NAME'	4	4	Fullword address of a control block in the MAD/I input/output support tables.
'TRANSFER POINT'	2	0	The first instruction at the transfer point.
'FIXED ARRAY' (—)	-	-	The component values, laid out by the array sequencing rule. The alignment and length are determined as in Section 3.1.2.1. There may also be an array dope vector, as described below.
'VARYING ARRAY' (—)	-	-	See 'FIXED ARRAY' above.
'COMPONENT STRUCTURE' (—)			The component values, laid out in the order declared. The alignment and length are determined as in Section 3.1.2.2. There may also be a dope vector, as described below.
'ALTERNATE' (—)	-	-	The alternative values, overlaid one "atop" the other. The alignment and length are determined as in Section 3.1.2.3.
'POINTER'	4	4	Fullword address of the item pointed to.
'ENTRY POINT'	2	0	The first instruction at the entry point.
'ENTRY NAME'	4	8	Fullword address of the entry point followed by the fullword address of the appropriate environment information.

Array Dope Vectors

An array dope vector is used to compute the displacement of a component within an array. The dope vector for an n-dimension array consists of the 3*n+1 items: n, L(1), U(1), M(1), ..., L(n), U(n), M(n), where each item is a fullword fixed-point number. n is the number of dimensions of the array, L(i) is the lower bound of the i-th subscript, U(i) is the upper bound of the i-th subscript, and M(i) is a multiplier used to compute the displacement of a component. The displacement of the component having subscripts (S(1), ..., S(n)) is computed as follows:

$$\text{displacement} = \sum_{1}^{n} [S(i) - L(i)] * M(i)$$

The upper bounds, U(i), are not used in this computation, but can be used to check subscript ranges.

For example, the declaration

'DECLARE' A 'FIXED ARRAY' (0...10, 5...20, 400) 'INTEGER'

produces the array dope vector (3, 0,10,25600, 5,20,1600, 1,400,4).

Component Structure Dope Vector

The dope vector for a component structure having n components consists of the n+1 items: n, D(1), ..., D(n). n is the number of components, D(i) is the displacement of the i-th component from the beginning of the component structure. Each item is a fullword fixed-point number.

For example, the declaration

'DECLARE' A 'COMPONENT STRUCTURE' ('INTEGER SHORT',
        'FLOATING LONG', 'BIT'(8))

produces the dope vector (3, 0, 8, 16).

Run-time Symbol Table

The format of the run-time symbol table is still in a state of flux, and is not defined here.

Section 15: Assembler Coding Feature

The assembler coding feature provides a minimal language facility for coding machine operations that cannot be expressed directly in MAD/I. Syntactically, the assembler coding feature consists of a compound statement in the MAD/I language. The scope of this statement consists of two parts: declarations and assembler-language statements. The machine code generated by the statement consists of the machine code specified by the assembler code in the statement scope, interspersed with compiler-generated machine code necessary to load base registers.

15.1 'ENTER ASSEMBLER CODE' Statement

The assembler coding feature statement is a compound statement which has a prefix of the form

          'ENTER ASSEMBLER CODE' ;

Note that only the long form of the compound statement is legal. The scope of the 'ENTER ASSEMBLER CODE' statement (abbreviated 'ENTASM') does not consist of MAD/I statements, but rather declarations peculiar to the 'ENTER ASSEMBLER CODE' statement followed by assembler code instructions. The individual declarations and assembler code instructions are separated by semicolons. The statement is terminated by the keyword 'END'.

15.1.1 Declarations

There are three declarations which can be specified in the scope of an 'ENTER ASSEMBLER CODE' statement. These are 'COVER', 'LABEL', and 'RESERVE'. Each declaration consists of one of the above three keywords followed by a list of identifiers and possibly constant symbols, separated by commas.

15.1.1.1 'COVER'

The 'COVER' declaration is used to guarantee that certain identifiers or constant symbols (not a-expressions) in the MAD/I program have base register coverage throughout the scope of the 'ENTER ASSEMBLER CODE' statement. 'COVER' should be used only for those identifiers and constant symbols for which compiler-generated load instructions preceding the assembler code instruction cannot be tolerated, because 'COVER' reserves registers for base coverage for each item in its list. One case in which 'COVER' should be used is for the identifiers and constant symbols referenced by the subject instruction of an EXECUTE instruction, since the insertion of load instructions

preceding the subject instruction (to acquire addressability)
would cause a load instruction, rather than the anticipated
instruction, to be executed. The following example causes up to
four general registers to be reserved for use as base registers,
one each for the two MAD/I identifiers QQSV and X, and one each
for the two constant symbols 15.3E-5 and 1:

                    'COVER' QQSV,X, 15.3E-5,1;

15.1.1.2 'LABEL'

The 'LABEL' declaration is used to declare that certain
identifiers will appear as labels within the scope of the 'ENTER
ASSEMBLER CODE' statement. The labels are defined by the
occurrence of a colon (:) followed by the label in what normally
would be called the label field of some assembler code
instruction. For example:

                    'LABEL' QQSV;
                       ●
                       ●
          :QQSV   L R3,X;

The scope of a 'LABEL' identifier is restricted to the 'ENTER
ASSEMBLER CODE' statement, and is independent of other
occurrences of the same symbol outside the statement.

15.1.1.3 'RESERVE'

The 'RESERVE' declaration is used to reserve general
registers for the use of the assembler language instructions
within the scope of the 'ENTER ASSEMBLER CODE' statement. Each
list item can either be an integer constant symbol, in which
case a specific general register is reserved, or an identifier,
in which case any available general register is reserved.
Identifiers representing registers are known only inside the
scope of the 'ENTER ASSEMBLER CODE' statement which defines
them, and are independent of the same symbols used outside of
that statement. It is best to mention specific registers first
and have the 'RESERVE' declaration precede any 'COVER'
declarations to insure that the register wanted has not already
been assigned to an identifier or as a base register. All
general registers other than registers 12 and 13 are available.
The compiler will feel free to use any registers which have not
been reserved. For example, the declaration

                    'RESERVE' 1,2,3,R1;

reserves general registers 1,2, and 3, plus one other arbitrary
general register whose designation will be R1.

15.1.2   Assembler Code Format

Assembler code instructions are written in much the same manner as in the assembler language, except that they are free-form and must be separated by semicolons. All the machine-instruction operation codes are valid, including the privileged operations, operations unique to the Model 67, and RFQ-ed instructions on the University of Michigan machine such as mixed floating-point, Swap Register, and the Search List instruction. None of the assembler instructions (such as EQU, ORG, DC, or USING) are valid.

The structure of the operands in the assembler language code is the same as in the assembler language (e.g., R,D(X,B)). However, the expressions which can be used as operands are much more restricted.

There are two kinds of "values" in the assembler code operand expressions: __absolute__ and __relocatable.__ __Relocatable__ values are __storage assignments.__ They are converted into base-displacement pairs when used as operands in assembler code instructions. __Absolute__ values, on the other hand, are equivalent to self-defining terms in the assembler language. They are used for register numbers, displacements, and immediate data.

The following can be used as expression operands:

1. Unsigned-integer constant symbols, which have the usual integer __absolute__ "value". For example, 10, 4, 0, and so forth.

2. Identifiers which have been 'RESERVE'ed, which have as their value the general register corresponding to them, which is an __absolute__ "value". For example, R1 following the declaration 'RESERVE' R1;

3. Constant symbols (not ∂-expressions) preceded by an equal sign (=), which have as their value the __relocatable__ storage assignment of the corresponding constant in the program. For example, =1, =10.5, ="FFF00000"X, and so forth.

4. Identifiers which appear as labels within the scope of the 'ENTER ASSEMBLER CODE' statement, which have as their value the __relocatable__ storage assignment of the corresponding assembler code instruction.

5. All other identifiers have as their value the __relocatable__ storage assignment of the corresponding identifier in the program.

The simplest of assembler code operands is one of the four
types of expression operands described above. These expression
operands can also be combined into more complicated expressions.
These expressions can then be used as assembler code operands.
The operators which can be used in forming expressions are
described below:

1. The addition operator (+), can be used to add together
the values of two operands. The result is <u>absolute</u> if both
operands are <u>absolute, relocatable</u> if either of the two
operands is <u>relocatable.</u> Meaningless values result if both
operands are <u>relocatable.</u> One   must   be   very   careful in
computing <u>relocatable</u> values, because the result may fall
outside of the area covered by the base register. For non-
structured modes, the entire storage assigned falls within
the base-area. For structured modes, only the first eight
bytes necessarily fall within the area.   Calculations
involving storage assignments of executable code   are
dangerous, because the compiler may begin a new base-area
at any point in an instruction sequence.

2. The prefix operator .LN.   accepts as an operand an
identifier within the program or a constant symbol preceded
by an equal sign, and returns as its result an <u>absolute</u>
"value" which is the compile-time length of the storage
assigned to the operand.

15.2 Interface Conventions

The assembler code instructions written in the scope of the 'ENTER ASSEMBLER CODE' statement of course are located in the larger environment of the code generated for all the statements in the program. Certain conventions are followed in the machine code generated by the MAD/I compiler and it is necessary for the user to be aware of some of them, although many steps have been taken to make these conventions as painless and transparent as possible.

15.2.1 Entry into the 'ENTER ASSEMBLER CODE' Statement

The 'ENTER ASSEMBLER CODE' statement can be entered in two ways, by "falling" into it under the normal sequencing rules of the language or by branching to (or calling) the label on the 'ENTER ASSEMBLER CODE' statement. In either case the execution of the assembler code within the statement begins with the first instruction. It is not possible to enter the assembler language code at any point other than its beginning.   The following operations are performed preceding the first assembler language statement:

1. If the 'ENTER ASSEMBLER CODE' statement has a label, all the usual code generated for a label is produced, including entry point code if the label is of 'ENTRY POINT' mode.

2. All unstored values in both the floating-point and general registers are stored into their respective variables.

3. All information concerning the contents of the registers ⁱ⸱
forgotten.   This essentially makes all the floating-po ⸱
registers and all the general registers other than 12 and available for use.

4.   The 'COVER' and 'RESERVE' declarations within the scope of the 'ENTER ASSEMBLER CODE' statement are processed in the order in which they appear. For each general register reserved the status of the register is changed to indicate that it cannot be used by the compiler for any purpose. For each MAD/I identifier or constant symbol covered, an available register is loaded with a base to cover the variable and its status is changed to indicate that it contains a base address and cannot be changed.

The result of these steps is that:

1. All floating-point registers are available for use by the assembler code.

2. All general registers (except 12 and 13) are available for

use by the assembler code. General registers are reserved
explicitly and implicitly by the 'RESERVE' and 'COVER'
declarations.

3. All general registers not reserved through 'RESERVE' or
'COVER' are available to the compiler for use as base registers.

4. General register 12 contains a base register used by the
compiler to maintain addressability. It covers the area called
%BASETAB, which contains the values put into base registers.

5. General register 13 contains the address of the save area to
be used for calling other subroutines. This contains a back
pointer to the save area provided by the program which called
the 'PROCEDURE' containing the 'ENTER ASSEMBLER CODE' statement.
In calling another subroutine, it is necessary to increment and
decrement the stack information used by MAD/I programs (the
stack contains the save areas). This will be shown in one of
the examples in Section 15.3.

6. All variable values are located in memory and must be
referenced from memory. The fact that a variable value might
also be in a register cannot be taken advantage of from the
assembler code.

15.2.2   Exit from the 'ENTER ASSEMBLER CODE' Statement

The 'ENTER ASSEMBLER CODE' statement can be left in three
ways: by "falling" out of the bottom following the normal
sequencing conventions, by branching to a label or 'ENTRY NAME'
variable, or by calling an 'ENTRY POINT' or 'ENTRY NAME'. In
each case there is no automatic storing of changed variable
values from the registers. It is entirely up to the user to
insure that all changed variable values are stored before the
'ENTER ASSEMBLER CODE' statement is exited. Furthermore, he
must follow all normal calling sequence conventions when calling
other subroutines, including the incrementing and decrementing
of stack information.

At the physical end of the scope of the 'ENTER ASSEMBLER
CODE' statement, all reserved registers are once again made
available to the compiler.

15.3   Examples

Below are several example 'ENTER ASSEMBLER CODE' statements. In each example, some operation is performed which cannot be adequately expressed in MAD/I. The examples attempt to show the correct balance between the use of MAD/I and the use of the assembler coding feature, with as much of the operation as possible being expressed directly in MAD/I. An attempt has been made to give useful examples that might indeed be used in actual programs. Each example contains line numbers (which are not a part of the actual code) and is followed by prose explaining each line of the example.

15.3.1   Generating a Standard OS Type (I) S Call

The MAD/I and standard OS type (I) S calling sequences differ in the structure of the parameter list. This difference in structure is transparent unless one is testing for variable-length parameter lists. In the standard parameter list, the end of the parameter list is indicated by having bit zero of the last parameter address set to one. In MAD/I, on the other hand, the number of parameters is specified in the word preceding the parameter list. This example calls the subroutine F passing three parameters, A, B, and C, following the standard calling conventions. This example also illustrates the incrementing of the stack address, which is necessary if the subroutine F causes a call on another subroutine written in MAD/I.

```
1          'DECLARE' F 'EXTERNAL' 'ENTRY POINT';
2          'DECLARE' PARS 'FIXED ARRAY'(3) 'POINTER';
3          'DECLARE' RTNCODE 'INTEGER';
4          PARS(1) := .PT. A; PARS(2) := .PT. B; PARS(3) :=
                 .PT. C;
5          PARS(3) := PARS(3) .V.  "80000000"X ;
6          'ENTER ASSEMBLER CODE';
7                  'RESERVE' 0,1,2,3,4,14,15;
8                  L        4,%STKADR;
9                  LA       14,72(0,13);
10                 LM       2,3,0(4);
11                 STM      13,14,0(4);
12                 LA       1,PARS;
13                 LA       15,F;
14                 BALR     14,15;
15                 STM      2,3,0(4);
16                 ST       15,RTNCODE;
17                 STE      0,RESULT;
18         'END';
```

1 declares F to be 'EXTERNAL' 'ENTRY POINT'. This is done implicitly in the normal MAD/I call (e.g., F.(A,B,C)).

2 declares PARS to be an array with components of 'POINTER'
mode. The parameter list for the standard OS type (I) S calling
sequence will be built in PARS.

3 declares RTNCODE to be of 'INTEGER' mode. The return code
from F will be stored here.

4 puts the addresses of A, B, and C into the parameter list.

5 sets bit zero of the address of C in the parameter list to
one, to conform to the standard OS conventions. The parameter
list is now complete.

6 begins the 'ENTER ASSEMBLER CODE' statement.

7 reserves general registers 0, 1, 2, 3, 4, 14, and 15 which are
used in a standard calling sequence and in saving and restoring
the stack status.

8 loads the address of the stack information into general
register 4. This is the first of the four instructions
necessary to increment the stack information to conform to MAD/I
stack conventions.

9 computes the current end of the stack.

10 saves the current two words of stack information in general
registers 2 and 3.

11 stores the two words of new stack information at the address
obtained in line 8.

12 loads the address of the parameter list into general register
1, to conform to standard calling sequence conventions.

13 loads the address of F into general register 15, to conform
to standard calling sequence conventions.

14 loads the return address into general register 14 and
branches to the entry point of F, according to standard calling
sequence conventions.

15 restores the two words of stack information saved at line 10.
This is the only instruction needed to decrement the stack.

16 stores the return code left by F from general register 15
into the variable RTNCODE.

17 stores the floating-point result returned by F from floating-
point register 0 into the variable RESULT.

18 terminates the 'ENTER ASSEMBLER CODE' statement.

15.3.2 Generating a Standard OS Type (I) R Call

The standard OS type (I) R call passes parameter values in the general registers rather than through a parameter list. This type of call cannot be directly generated by any higher level language, and yet it is useful because many MTS system subroutines follow this calling convention. This example calls the MTS system subroutine GETFD, which acquires a file or device given the address of its EBCDIC name in general register one, and returns the address of a control block called a FDUB in general register zero. This address can be used in further I/O operations on the file or device. In this example, the EBCDIC name is assumed to be the value of the variable NAME and the FDUB address is stored in the variable FDUB. Note that the stack information is <u>not</u> incremented. This is not necessary because GETFD will not call any MAD/I procedure.

```
1           'DECLARE' GETFD 'EXTERNAL' 'ENTRY POINT';
2           'DECLARE' FDUB 'INTEGER';
3           'DECLARE' RTNCODE 'INTEGER';
4           'DECLARE' NAME 'CHARACTER'(80) ;
5           'ENTER ASSEMBLER CODE';
6                   'RESERVE' 0,1,14,15;
7                   LA        1,NAME;
8                   LA        15,GETFD;
9                   BALR      14,15;
10                  ST        15,RTNCODE;
11                  ST        0,FDUB;
12          'END';
```

1 declares GETFD to be 'EXTERNAL' 'ENTRY POINT'. This is done implicitly in the normal MAD/I call (e.g., GETFD.(NAME)).

2 declares FDUB to be of 'INTEGER' mode. It actually does not matter what mode it is, so long as it has length 4 and alignment 4.

3 declares RTNCODE to be of 'INTEGER' mode. The return code from GETFD will be stored here.

4 declares NAME to be of 'CHARACTER'(80) mode. The name of the file or device followed by at least one blank is assumed to be here.

5 begins the 'ENTER ASSEMBLER CODE' statement.

6 reserves general registers 0, 1, 14, and 15 which are used in the calling sequence.

7 loads the address of the EBCDIC name into general register 1.

8 loads the address of the entry point to GETFD into general

register 15.

9 loads the return address into general register 14 and branches
to the entry point of GETFD.

10 stores the return code left by GETFD from general register 15
into the variable RTNCODE.

11 stores the FDUB address returned by GETFD from general
register 0 into the variable FDUB.

12 terminates the 'ENTER ASSEMBLER CODE' statement.

15.3.3 Translating Lower-case Characters to Upper Case

The System/360 has a powerful instruction (translate)
useful for translating from one character set encoding to
another. The desired translation is defined by a 256-byte
translate table. MTS has several translate tables which can be
referenced as external symbols to perform common translations.
One of these is CASECONV, which converts all lower-case
alphabetic characters to upper-case alphabetic characters. The
following example converts any lower-case characters in the
variable STRING to upper-case characters.

```
1          'DECLARE' CASECONV 'EXTERNAL' 'FIXED ARRAY'(256)
                     'CHARACTER'(1) ;
2          'DECLARE' STRING 'CHARACTER'(80) ;
3          'ENTER ASSEMBLER CODE';
4                   TR          STRING(80),CASECONV;
5          'END';
```

1 declares CASECONV to be of 'EXTERNAL' storage class. The
remainder of the declaration is not important unless CASECONV is
referenced in normal MAD/I code.

2 declares STRING, the character string to be translated, to be
of 'CHARACTER'(80) mode.

3 begins the 'ENTER ASSEMBLER CODE' statement.

4 translates the characters of STRING using the translate table
CASECONV.

5 terminates the 'ENTER ASSEMBLER CODE' statement.

15.3.4 Converting an 'INTEGER' to Hexadecimal Characters

This example translates the 'INTEGER' variable NUMBER into a string of hexadecimal characters in the 'CHARACTER'(8) variable HEXOUT.

```
1        'DECLARE' NUMBER 'INTEGER' 'LENGTH'(5) ;
2        'DECLARE' HEXOUT 'CHARACTER'(8) ;
3        'DECLARE' WORK 'CHARACTER'(9) ;
4        'DECLARE' TAELE 'FIXED ARRAY'(256) 'CHARACTER'(1) ;
5        'PRESET' TABLE(241) := "0", "1", "2", "3", "4", "5",
                 "6", "7", "8", "9", "A", "B", "C", "D", "E",
                 "F";
6        'ENTER ASSEMBLER CODE';
7                UNPK        WORK(9),NUMBER(5);
8                TR          WORK(8),TABLE;
9                MVC         HEXOUT(8),WORK;
10       'END';
```

1 declares NUMBER to be of 'INTEGER' 'LENGTH'(5) mode. This causes five bytes to be allocated to NUMBER, the first four containing its value and the last being unused. This unused byte is needed because of the idiosyncrasies of the UNPK instruction with the low-order byte as pertains to this usage of it.

2 declares HEXOUT to be of 'CHARACTER'(8) mode. The hexadecimal character string result is left here.

3 declares WORK to be of 'CHARACTER'(9) mode. This is a wcrk area used during the conversion.

4 declares TABLE to be a 'FIXED ARRAY' of 'CHARACTER'(1) components. This is the translate table which is referenced at line 8.

5 presets the translate table appropriately.

6 begins the 'ENTER ASSEMBLER CODE' statement.

7 unpacks the four bytes of the value of NUMBER into the first eight bytes of WORK. The four-bit values 0...F are expanded into the eight-bit values F0...FF. The last byte of both NUMBER and WORK are treated as the sign and low-order digit by the UNPK instruction and are ignored by this algorithm.

8 translates the eight bytes of the result from the values F0...FF to the appropriate EBCDIC character representation.

9 moves this result into the variable HEXOUT.

10 terminates the 'ENTER ASSEMBLER CODE' statement.

15.3.5. Moving an Arbitrary Number of Characters

This example moves n characters from A(i)...A(i+n-1) to B(j)...B(j+n-1) , where A and B are both fixed arrays of 'CHARACTER'(1) elements. It assumes that $1 \leq n \leq 256$.

```
1          'DECLARE' (A,B) 'FIXED ARRAY'(32768) 'CHARACTER'(1);
2          'DECLARE' (I,J,N) 'INTEGER';
3          'DECLARE' (PTA,PTB) 'POINTER';
4          PTA := .PT.  A(I);
5          PTB := .PT.  B(J);
6          'ENTER ASSEMBLER CODE';
7                    'RESERVE' RLEN, RA, RB;
8                    'LABEL'   SKIP, EXTHIS;
9                    B         SKIP;
10 :EXTHIS           MVC       0(0,RB),0(RA);
11 :SKIP             L         RLEN,N;
12                   L         RA,PTA;
13                   L         RB,PTB;
14                   BCTR      RLEN,0;
15                   EX        RLEN,EXTHIS;
16         'END';
```

1 declares A and B to be of 'FIXED ARRAY'(32768) 'CHARACTER'(1) mode.

2 declares I, J, and N to be of 'INTEGER' mode.

3 declares PTA and PTB to be of 'POINTER' mode.

4 puts the address of A(i) into PTA.

5 puts the address of B(j) into PTB.

6 begins the 'ENTER ASSEMBLER CODE' statement.

7 reserves three general-purpose registers named RLEN, RA, and RB for use in this assembler code section.

8 declares two local labels, SKIP and EXTHIS.

9 branches to the next instruction to be executed. This transfers around line 10 which will be the subject instruction of an execute instruction.

10 is the subject instruction of the execute instruction of line 15. It performs the actual move.

11 loads the number of characters to be moved into register RLEN.

12 loads the address of A(i) into register RA.

13 loads the address of B(j) into register RB.

14 subtracts one from the length, for the MVC instruction.

15 executes the MVC instruction to move the n characters.

16 terminates the 'ENTER ASSEMBLER CODE' statement.

15.3.6   Reading from SCARDS into a 'VARYING CHARACTER' Variable

This example reads a variable-length input record via the
MTS subroutine SCARDS and then sets up a 'VARYING CHARACTER'
variable so that it is the contents of the record that has been
read.   It assumes that the record read will have a length
greater than zero and less than 256.


```
1            'DECLARE' STRING 'VARYING CHARACTER'(255);
2            'DECLARE' INAREA 'CHARACTER'(255);
3            'DECLARE' LEN 'INTEGER SHORT';
4            'DECLARE' LINNUMB 'INTEGER';
5            SCARDS.(INAREA,LEN,0,LINNUMB);
6            'ENTER ASSEMBLER CODE';
7                      'RESERVE' RLEN;
8                      'COVER'   STRING,INAREA;
9                      'LABEL'   SKIP,EXTHIS;
10           B         SKIP;
11   :EXTHIS MVC       STRING+2(0),INAREA;
12   :SKIP   LH        RLE..,LEN;
13           STH       RLEN,STRING;
14           BCTR      RLEN,0;
15           EX        RLEN,EXTHIS;
16      'END';
```

1 declares STRING to be of 'VARYING CHARACTER'(255) mode.

2 declares INAREA to be of 'CHARACTER'(255) mode.

3 declares LEN to be of 'INTEGER SHORT' mode.

4 declares LINNUMB to be of 'INTEGER' mode.

5 reads the next record into INAREA, putting its length into LEN
and its line number into LINNUMB.

6 begins the 'ENTER ASSEMBLER CODE' statement.

7 reserves a general-purpose register and names it RLEN.

8 guarantees that the variables STRING and INAREA have base-register coverage throughout the 'ENTER ASSEMBLER CODE' statement. This is necessary because these variables are referenced by the MVC instruction of line 11 which is the subject of the EX instruction of line 15.

9 declares two local labels, SKIP and EXTHIS.

10 branches to the next instruction to be executed. This transfers around line 11 which is the subject instruction of the execute instruction on line 15.

11 is the subject instruction of the execute instruction on line 15. It moves the contents of the string from the input record in the variable INAREA into the proper location within the variable STRING.

12 loads the length of the string into RLEN.

13 stores the length of the string into the proper area in STRING.

14 subtracts one from the length, for the MVC instruction.

15 executes the MVC instruction to move the string into STRING.

16 terminates the 'ENTER ASSEMBLER CODE' statement.

APPENDICES

Appendix A: Syntax Notation

This notation is used to describe the syntax of MAD/I. It
does not describe the meaning of language elements but only the
syntax, e.g., the order of elements, punctuation, and options
that may occur.   Note that this syntax notation is used for
describing MAD/I but is not itself part of the MAD/I language.

The following describes the syntax notation:

Notation Variable

A notation variable is a name for a construction in the
MAD/I language. It may be formed by:

1) Lower-case letters and decimal digits and it must begin
    with a letter.

2) A combination of lower- and upper-case letters and decimal
    digits.   There must be at least one portion in all lower-
    case. Each portion is joined to the adjacent portions with
    a hyphen.

Examples: expression
                   identifier
                   procedure-call
                   VALUE-statement

All notation variables are defined either in terms of this
syntax notation or in terms of English. If a notation variable
is defined with this syntax notation, the variable occurs to the
left of the definition operator = and the definition occurs to
the right.

Notation Constant

A notation constant stands for the literal occurrence of
the characters composing the constant. A notation constant
consists of upper-case letters, digits, and special characters.
It may not consist of any lower-case letters.

Example: 'LENGTH'

This denotes the literal occurrence of the characters 'LENGTH' .

Concatenation

When two or more notation elements are written adjacent, they denote an occurrence of the first element followed by an occurrence of the second element, and so on. Blank spaces between notation elements have no significance.

Example: 'LENGTH' (integer)

This denotes an occurrence of the literal characters 'LENGTH' followed by a literal left-parenthesis, followed by a construction denoted by the notation variable "integer", followed by a literal right-parenthesis.

Alternation |

The vertical bar | is used to indicate that a choice is to be made.

Example:

      storage-class = 'BASED' | 'STATIC' | 'AUTOMATIC'

This means that "storage-class" is defined to be either 'BASED' or 'STATIC' or 'AUTOMATIC'. Alternation has lower precedence than concatenation; e.g., x | y z means x | {y z} .

Grouping { }

The braces { } may be used to denote grouping among notation elements.

Example:

      array = { 'FIXED ARRAY' | 'VARIABLE ARRAY' } dimension

This is equivalent to

   array = 'FIXED ARRAY' dimension | 'VARIABLE ARRAY' dimension

Optionality [ ]

The square brackets [ ] are used to indicate that something is optional. Whatever is enclosed in square brackets either may appear or may not appear. In addition, the brackets imply a grouping of the notational elements enclosed within them.

Example: lower-bound = [-] integer

This is equivalent to

$$\text{lower-bound} = - \text{integer} \mid \text{integer}$$

Repetition

The notation keyword <u>list</u> (which must always be underlined) may be used to represent a sequence of items. It may be followed by either one or two notation expressions. If it has one argument, it stands for that argument occurring one or more times in succession.

I.e.,

<u>list</u> x is equivalent to x | xx | xxx | ...

If <u>list</u> is used with two arguments, it stands for a sequence of one or more of the second argument separated by occurrences of the first argument. I.e.,

<u>list</u> x y is equivalent to y [<u>list</u> {x y}]

Example:

<u>list</u> , label

is equivalent to:

label | label,label | label,label,label | ... .

The following precedence holds:

<u>list</u> x y is equivalent to { <u>list</u> x y }

and is not equivalent to { <u>list</u> x } y ;

<u>list</u> x y z is equivalent to { <u>list</u> x y } z

rather than <u>list</u> x { y z } .

Order Independence

The # notation is used where order is not important, i.e.,

x # y is equivalent to x y | y x

The # notation has higher precedence than either alternation or concatenation; e.g.,

a b # c | d is equivalent to a { b # c } | d .

Appendix B: Summary of Pre-defined Symbols

Symbol	Abbreviation	Section(s)

Primed symbols

Symbol	Abbreviation	Section(s)
'ACCESSIBLE'	'ACC'	3.3
'ALIGN'		3.2.2
'ALLOCATE'		5.13
'ALTERNATE'	'ALT'	3.1.2.3
'AUTOMATIC'		3.4.2
'BASED'		3.4.3
'BEGIN'		5.10
'BIT'		3.1.1.6
'BLOCK'		5.10, 7.1
'BOOLEAN'	'BOOL'	3.1.1.7
'CHARACTER'	'C'	3.1.1.8
'CLOSE'		6.9.1
'COMPONENT STRUCTURE'	'CS'	3.1.2.2
'COVER'		15.1.1
'DATA SET'		6.3.2
'DEALLOCATE'		5.13
'DECLARE'	'DCL'	3.6.1, 5.9
'DECLARE CSECT'		5.12
'DECLARE DEFAULT'	'DCLD'	3.7.2, 5.9
'DECLARE PSECT'		5.12
'DEFAULT'		3.7.2
'ECHO'		6.4.6
'END'		5.0, 5.10
'END FOR'		5.4, 5.5
'END IF'		5.3
'END OF FILE'	'EOF'	6.4.2
'END OF VOLUME'	'EOV'	6.4.3
'END PROCEDURE'		5.7
'END SUBSTITUTE'	'ENDSUB'	8.1
'END VALUE' .		5.6
'ENTER ASSEMBLER CODE'		15.1
'ENTER FACILITY'		9
'ENTRIES'		6.3.4
'ENTRY NAME'	'EN'	3.1.2.6
'ENTRY POINT'	'EP'	3.1.2.5
'ERROR'		6.4.4
'ERROR EXIT'		4.2.7
'EXTERNAL'	'EXT'	3.3
'FALSE'		3.1.1.7, 2.2.2
'FILE NAME'		3.1.1.10, 6.1
'FIXED ARRAY'	'FA'	3.1.2.1.1
'FLOATING'	'F'	3.1.1.3
'FLOATING LONG'	'FL'	3.1.1.4
'FLOATING SHORT'	'FS'	3.1.1.3
'FOR'		5.4

'FOR VALUES'		5.5
'FORMAT'		6.5.1
'GLOBAL'		3.3
'GO TO'		5.2
'IF'		5.3
'INCLUDE'		8.2
'INTEGER'	'I'	3.1.1.2
'INTEGER LONG'	'IL'	3.1.1.2
'INTEGER SHORT'	'IS'	3.1.1.1
'LABEL'		15.1.1
'LAST LENGTH'		6.5.4
'LAST LINE'		6.5.3
'LENGTH'		3.2.1
'LINE'		6.5.2
'LIST'		6.7.5
'MAX LENGTH'		6.4.5
'NEW'		3.3
'NOT NEW'		3.3
'NULL C'		3.1.1.8
'NULL EN'		3.1.2.6
'NULL PT'		3.1.2.4
'NULL VC'		3.1.1.9
'OPEN'		6.9.1
'OR ELSE'	'ELSE'	5.3
'OR IF'		5.3
'PACKED'	'P'	3.1.1.5
'POINTER'	'PT'	3.1.2.4
'POP SUBSTITUTE'		8.1
'PRESET'		5.11
'PROCEDURE'	'PROC'	5.7, 7.1
'READ'		6.9.3, 6.9.4
'READ DATA'		6.9.2
'READ UNCONVERTED'		6.9.5
'REDIMENSION'		5.14
'RESERVE'		15.1.1
'RETURN'		5.7.3
'RETURN TO'		5.7.3
'SAVE CODE'		4.2.7
'STATIC'		3.4.1
'STRING DATA SET'		6.3.3
'SUBSTITUTE'		8.1
'TO'		5.14
'TRANSFER POINT'		3.1.1.11
'TRUE'		3.1.1.7, 2.2.2
'UNIT'		6.3.1
'VALUE'		5.6
'VARYING ARRAY'	'VA'	3.1.2.1.2
'VARYING CHARACTER'	'VC'	3.1.1.9
'WITH'		5.2
'WRITE'		6.9.3, 6.9.4
'WRITE DATA'		6.9.2
'WRITE UNCONVERTED'		6.9.5

Dotted_symbols

.A.	4.2.4
.ABS.	4.2.1
.ALLOC.	4.2.10
.AND.	4.2.3
.AS.	3.9
.ASTYPE.	3.9
.ASTYPEOF.	3.9
.CONCAT.	4.2.5
.CONV.	4.2.5
.ENCON.	4.2.10
.EQV.	4.2.3
.EV.	4.2.4
.EXOR.	4.2.3
.IND.	4.2.10
.LN.	4.2.10
.LS.	4.2.4
.LSA.	4.2.4
.N.	4.2.4
.NE.	4.2.2
.NEG.	2.2.4, 4.2.1
.NOT.	4.2.3
.OR.	4.2.3
.PT.	4.2.10
.PTCON.	4.2.10
.REM.	4.2.1
.RS.	4.2.4
.RSA.	4.2.4
.TAG.	4.2.6
.THEN.	4.2.3
.V.	4.2.4

Special symbols (see also Section 2.1.7)

Symbol	Name	Section

Punctuation symbols:

(	left-parenthesis	
)	right-parenthesis	
,	comma	
;	semicolon	
:	colon	2.2.1.2, 3.7.1, 5.0
...	ellipsis	3.1.2, 6.7.1
#	pound-sign	4.2.6, 5.7.2

Operators: (see also Section 4.3 for precedences)

+	plus	4.2.1
-	minus	4.2.1
*	asterisk	4.2.1
/	slash	4.2.1
**	double-asterisk	4.2.1
=	equal-sign	4.2.2
<	less-than	4.2.2
>	greater-than	4.2.2
¬=	not-equal	4.2.2
<=	less-than-or-equal	4.2.2
>=	greater-than-or-equal	4.2.5
@	at-sign	3.6.3, 3.8
$	dollar-sign	4.2.6
.	dot, period	4.2.7, 3.7.1
¬	not-sign	4.2.3
&	ampersand	4.2.3
I	vertical bar	4.2.3
II	double-bar	4.2.5
:=	colon-equals, assignment	4.2.9

Also, the two-character sequence << is reserved for use as a comment delimiter.

Appendix C: Current Restrictions & Possible Extensions

Implementation Restrictions

The following are current implementation restrictions. They are coded by section number within the manual.

Section	Restriction
2.1.4.3	Pointer-constant symbols are not yet supported. The same effect can be obtained through the .PTCON. operator. See Section 4.2.10.
2.1.4.4	Entry-name-constant symbols are not yet supported. The same effect can be obtained through the .ENCON. operator. See Section 4.2.10.
2.2.5	The "$" operator is not yet supported. Components can be accessed by using the component name as if it were a subscript; e.g., COMPLXN(@REAL) instead of COMPLXN $ @REAL).
3.1.2.2	See the restriction under Section 2.2.5 concerning the "$" operator.
3.4.2	Automatic storage class is not yet supported.
4.2.1	The operator-mode combinations which involve 'BIT' mode as both the first and second operand modes are not yet implemented.
4.2.4	The bit-string operations are not yet defined for 'BIT' mode.
4.2.5	The concatenation operation currently is not implemented for 'VARYING CHARACTER' mode.
4.2.6	The "$" operator is not yet implemented.
4.2.6	Substring selection is not yet implemented for 'VARYING CHARACTER' mode.
4.2.7	The phrase keywords 'ERROR EXIT' and 'SAVE CODE' are not yet implemented. Instead, the variable %RTNCODE contains the value of the last return code from the last procedure called. %RTNCODE should be interrogated as soon as possible, since compiler-generated subroutine calls (for I/O, subscription, etc.) also modify its value.

5.4   The 'FOR' statement cannot appear as an embedded
            statement in the prefix part of another 'FOR'
            statement or 'FOR VALUES' statement.

5.6   The prefix 'VALUE' V := E is not yet implemented. The
            same effect can be obtained by: 'VALUE' V; V := E ....

5.7.2   The declaration of a formal parameter with an "array-
            suffix" in which the "bounds" entries are the special
            symbol # is not yet implemented.

5.14   The 'REDIMENSION' statement is not yet implemented.

6.7.1   The elements of an array which are referenced in a
            block-element must have a length equal to the "aligned
            length" of an array component. See Section 3.1.2.1
            for a discussion of aligned length. Hence given the
            following declarations, only A can be referenced in a
            block-element:

            'DECLARE' A 'FA'(10,15) 'I',
                  B 'FA'(10,15) 'ALIGN'(8) 'I',
                  C 'FA'(10,15) 'LENGTH'(7) 'I'

6.7.2   Array expressions are not yet supported in data-lists.

6.7.3   Component-structure expressions are not yet supported
            in data-lists.

6.9.1   The 'OPEN' and 'CLOSE' statements are not yet
            implemented.

6.9.2   The data-directed input/output statements ('READ DATA'
            and 'WRITE DATA') are not yet implemented.

6.9.3   List-directed input/output is not yet implemented.

6.9.4   Only a subset of format-directed input/output is
            currently implemented. The I/O-spec-list must always
            be specified. Its elements must be specified in
            positional form. The first element is taken as the
            format and is mandatory. The second element is
            optional and is interpreted in the following ways:

            (1) If it is absent, the logical I/O unit SCARDS is
                used for input; SPRINT for output.
            (2) If the first byte is zero, it is taken as an
                integer unit specification; that is, a
                specification of logical I/O unit 0 through 9 or
                a FDUB.
            (3) In all other cases, it is taken as an FDname, and
                must be terminated by a blank character.

No other input/output specifications are allowed.

6.9.5   Unconverted input/output is not yet implemented.

7.1   The outermost block must be a procedure block;
           programs written as compound-statement blocks may not
           compile.

15.1.2   Identifiers used as operands as discussed in (5) must
           belong to the outermost block of the program.

Possible Extensions

        The following are extensions to the MAD/I language and
MAD/I compiler which are anticipated as future developments.

Section   Expected Extensions

3.1.1.8   The maximum number of characters allowed in
           'CHARACTER' mode values is expected to be increased to
           32767.   Lengths   greater   than   256   will   cause
           subroutines to be called when used as operands to most
           operations.

4.2.10   Operations comparable to the PL/I built-in functions
           INDEX, TRANSLATE, and VERIFY are contemplated.

6.7.1   Block-elements are expected to be defined across
           components within all the structured modes, not just
           the array modes.

7.3   Recursive procedures are contemplated; such procedures
           would   require   the declaration of a 'RECURSIVE'
           attribute.